Recommender System for News Feed

# Description

There is an increasingly large amount of content produced by many entities that can be of interest to our users. However, users can only consume a tiny fraction of this content that is relevant to them, fresh and that reaches them at the right time. Moreover, this content may come in many forms: news articles, blog posts, pictures, videos, status updates from other entities, etc. We need a system to recommend *good* content to our users that they enjoy and benefits them.

# Requirements and Goals

Requirements:

* The system should work at Facebook’s scale.
* It should be accessible by smartphone and give locally meaningful recommendations.
* Availability: highly available, anywhere at any time, with minimal latency.
* The content should be engaging and enjoyable.

Goals:

* The north metric should be user satisfaction as measured by online/offline feedback, likes and hearts.
* We should maximize revisit rate.
* We should maximize session duration.
* We should maximize re-shares.
* We should minimize bad press and escalations. That is, our recommender system should not be a reason for the news media to criticize Facebook.

# Capacity Estimation and Constraints

Let’s assume we have about 2B daily active users in almost every country in the world, most of them accessing using a smartphone, visiting an average of twice per day, scrolling through 100 pieces of content and interacting with 10 of them.

For efficiency and due to policy constraints, this data is distributed across servers in many regions. Given the large amounts of data, spread in many countries, in so many different languages, we would opt for global models trained on local data.

We have 50 million news WW, each user is connected to 200 friends in average, 5 groups, and potentially receives 100 daily friends and group updates.

# System APIs

There are 2 types of APIs we would need to support:

1. get\_recommendations(entity\_id, content\_types, start\_time, end\_time, sort\_by, recommender\_system\_id)
   1. entity\_id: the entity (e.g. user, group) for which we want to produce the recommendations.
   2. Content\_types: a list of content types that we want to recommend. For example, [news, blogs, photos, videos]. Each of these content types will have a field to specify the ratio to total content that this type should have. For example, it would allow us to specify that 10% of the content should be news articles, 20% should be photos from friends, 20% should be videos, etc.
   3. start\_time and end\_time: timestamps to filter content within that range. This will be useful to enable extended content search.
   4. Sort\_by: function name to sort by, e.g. popularity, relevance, trending, etc.
   5. Recommender\_system\_id: Overall system that should serve the recommendations. This will be useful to do small-scale experiments such as Eat Your Own Dog Food Sessions, system demonstrations, explicit user opt-ins, etc.
2. model\_update(model\_id, \*\*config):
   1. This is a model management API that allows us to dial-up or dial-down models so that there is a larger or smaller percent of users impacted, replacing model images, allocating more machines, increasing/decreasing batch size which is the time the model waits to fill the batch before executing, increasing latency in exchange of efficiency.

# Database Design

Entities:

| entity\_id | Big int |
| --- | --- |
| entity\_type | String: user, group, network |
| Creation\_date | timestamp |
| Update\_date | timestamp |
| is\_active | bool |
| region\_id | int |
| country\_id | int |
| latitude | double |
| longitude | double |
| gender | string |
| date\_of\_birth | timestamp |
| country\_of\_birth | string |
| city\_of\_birth | string |
| primary\_language | string |

Content

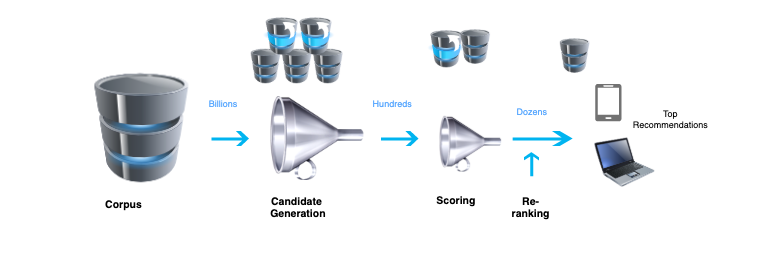
| content\_id |  |
| --- | --- |
| creator\_id |  |
| content\_type |  |
| category\_code |  |
| subcategory\_code |  |
| label\_ids |  |
| creation\_date |  |
| update\_date |  |
| removal\_date |  |
| trust\_score |  |
| trust\_code |  |
| is\_abusive |  |
| abuse\_code |  |
| title |  |
| description |  |
| snippet |  |
| url |  |
| image\_id |  |
| content\_key |  |
| Latitude |  |
| Longitude |  |

User Interactions

| user\_id |  |
| --- | --- |
| content\_id |  |
| Glance view time |  |
| liked |  |
| reshared |  |
| visit\_date |  |

# High Level Design

For each user, we cannot possibly scan billions of pieces of content to produce the optimal recommendations. Instead, we adopt a multi-stage strategy that is typical in the literature of recommender systems:



First, we index all content so that we can do constant time look-ups in single-digit milliseconds. Then, we run a sequence of three stages where each stage could have pre-computed intermediate results to speed up the process.

1. The candidate generation stage uses simple and fast models that are able to generate hundreds of possible items for each query (user) and context (user history, access location, device info, time of the day, etc). Examples of queries are user information (e.g. status, profile, stated interests) or group information (group title, description); examples of contexts are the time of the day, day of the week or region where the user is.
2. Then we use more sophisticated models to score the candidate pieces of content using more information about the entity, the content and the context such as the most recently clicked pieces of content, etc. With this, we produce a few dozen pieces of content that we would like to display.
3. Finally, we can use a more expensive procedure to re-rank these few pieces of content so that they obey more global and desirable properties such as diversity, fairness and freshness.

## Candidate Generation

There are two common approaches to candidate generation, namely content-based filtering and collaborative filtering. They diverge in the type of information they use to select candidates:

* **Content-Based Filtering:** Content-based filtering involves recommending items based on the attributes of the items themselves. The system recommends items similar to what a user has liked in the past. For instance, if the user recently read a post about a scientific development in COVID-19, then we could recommend a popular post on COVID-19 treatment.
* **Collaborative Filtering:** Collaborative filtering relies on the user-item interaction and relies on the concept that similar users like similar things e.g. Customers who bought this item also bought this. For example, if other users in her group read some post, we could recommend this post to her too.

## Scoring and Ranking

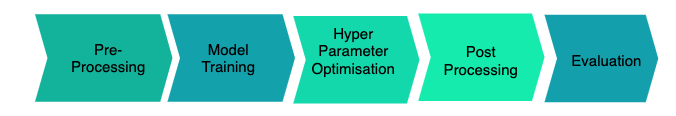
Broadly, there are three strategies to do ranking: point-wise, pairwise and listwise ranking. The simplest and most popular choice is point-wise ranking, which scores a piece of content for a given user independently on other pieces of content or users. Neural networks are widely used for this task, which are able to both learn representations of users and content, and integrate explicit signals from these users and content such as user demographics or content types.

## Re-ranking

At the re-ranking stage, we can optimize the feed to follow desirable properties such as diversity, fairness and freshness. For example, we can decide the order of widgets or categories we want to display in the vertical order to maximize content category diversity, e.g. first a row of news, then a row of images or moments, then a row of videos from your group, etc. Within each row, we would order those items according to freshness and fairness.

# Low Level Design

Each of the stages described in the high level design have their own development stages, typically consisting in:



In this section we will discuss different options for Candidate Generation, Scoring/Ranking and Re-ranking, together with their pros and cons.

## Candidate Generation

There are two main strategies for Candidate Generation, namely content-base filtering and collaborative filtering. These strategies are typically combined into a [hybrid system](https://en.wikipedia.org/wiki/Recommender_system#Hybrid_recommender_systems). Some combination methods are:

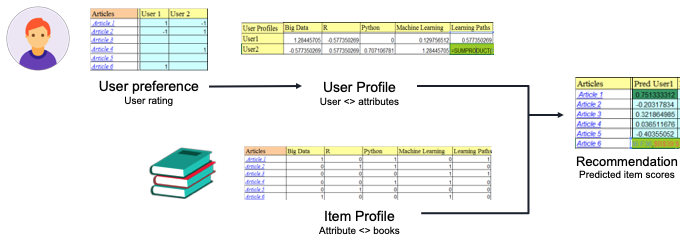
* Weighting: each source or each item within each source has a numeric score which is combined with the item scores from other candidate generators.
* Use different candidate generators depending on the stage of each user. For instance, if the user is new, then we would rely more on content-based filtering. When we know a lot about the user, then we can rely more on collaborative filtering.
* Scores or features from each candidate generator are concatenated and used as input features for a new candidate generator.

### Content-Based Filtering

In content-based filtering, we use similarities between currently or previously seen items to unseen items, to power experiences such as “similar items you might be interested in”. To realize this model, we need a way to featurize or embed each piece of content and a fast method to retrieve the k-nearest neighbors:

* Typical featurizations for a News Feed would include: the type of content (news article, video post, etc.), genre of news, bag of words of titles, descriptions and body of text, its TF-IDF counterpart, object detection in images, **location**, content creator or creator’s tags.
* Content embedding routines could include the item embeddings in matrix factorization (see collaborative filtering section), node embeddings in graph neural networks or any other embedding learned as part of a neural network training process.
* K-nearest neighbors need to scale to millions of items. However, we cannot afford N^2 comparisons to find item similarities for each item. One strategy is to use [Random Projections](https://en.wikipedia.org/wiki/Random_projection) or [Locality-Sensitive Hashing](https://en.wikipedia.org/wiki/Locality-sensitive_hashing) to find hash collisions and retrieve those collisions (i.e. other pieces of news or photos) in constant time.

We could do some personalization on this candidate generation by including information about the user profile. The idea is to calculate the weighted average of feature vectors of each content the user has interacted with in the past X days. The weights for each feature vector could be the percentage of time the user spent on each piece of content. The result will be a user preference vector that we can use to calculate similarities with other content items. However, this will give *average* user preferences which might be problematic if the user has interests in opposing sides of the spectrum. This diagram illustrates the process:



Pros:

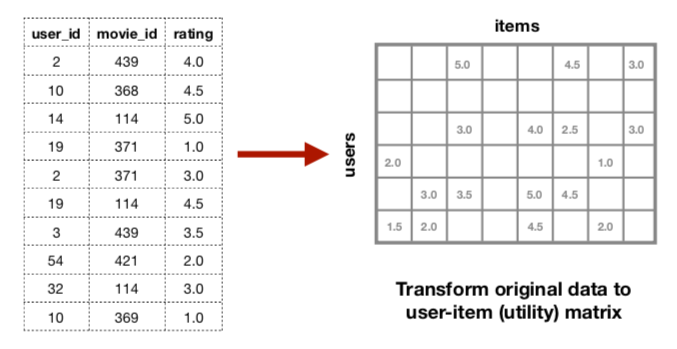
* It doesn't have a cold-start problem on items because it works through attributes or tags of the **content**, such as actors, genres or directors, so that new movies can be recommended right away.
* Fast and scalable; doesn’t depend on other users. Specific to the user in the personalization setting.
* Easy to interpret and explain the recommendations to our users. For example, we can say “because you read XYZ article, we recommend you these other articles”.

Cons:

* It doesn’t help users to discover new interests or go out of their information bubble.
* The recommended items might be too similar to what the user already consumed, thus not being too engaging.
* Requires hand-engineered features. If we don’t have enough information to extract good features from, then the item will never surface.
* Cold-start problem for absolute new users.

### Collaborative Filtering

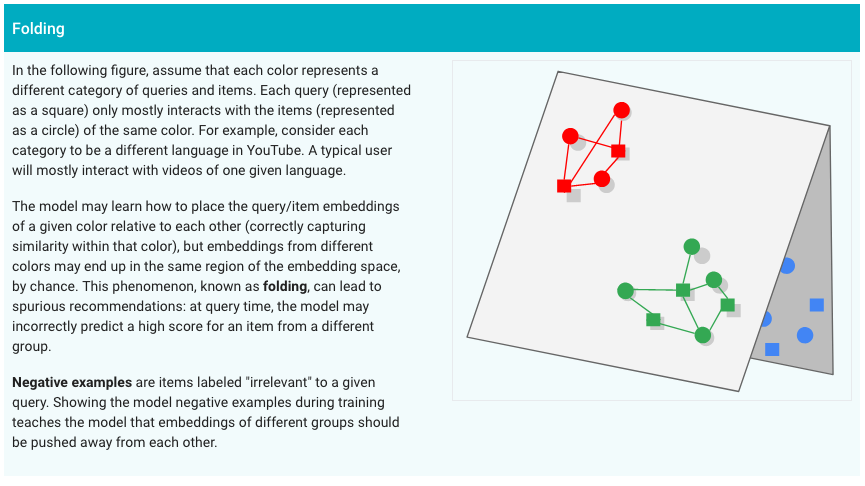
In Collaborative Filtering, we use similarities between users to recommend items that other users liked. For this purpose, we need to use past user-item interactions to predict positive interactions for other similar users. These user-item interactions are typically user-movie ratings, user-article “likes” or “re-shares”, e.g. when a user likes an article and s/he saves it for later, or likes it, or re-shares it with his/her network. These user-item interactions are saved in the form of a Feedback matrix (or Utility matrix) A, where each row is a user and each column is an item (i.e., a news article or blog post). This matrix is very sparse with as many rows as users and as many columns as items. If we are working with ratings, then these ratings can be normalized per user to account for systematically positive or negative users.



Let A\_ij be the rating of user i for item j. The objective is to f[actorize this matrix](https://developers.google.com/machine-learning/recommendation/collaborative/matrix) into a user matrix U (dimension *u* x *d*) and Item matrix I (dimension *i* x *d*) where *u* is the number of users, *i* is the number items, and *d* is the dimensionality of a latent space, e.g. 300 or 500. The objective is then to estimate the matrices U and I such that they minimize the following loss function:

Loss = \sum (A\_ij - U\_i \* I\_j)^2

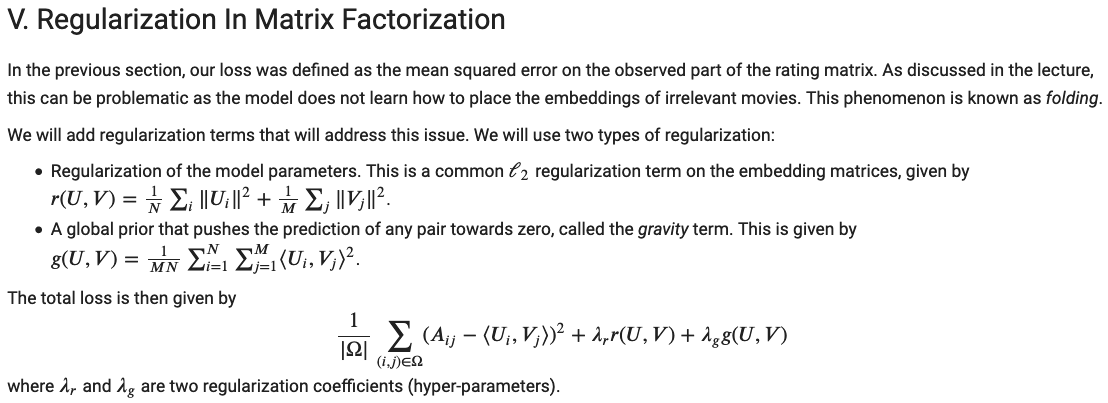
where U\_i is the *d*-dimensional row of matrix U that represents user i and I\_j is the d-dimensional row of matrix I that represents item j. Their dot product should result in the rating that the user i gave to item j in the past and hopefully generalize to unseen user-item pairs. This loss function can be optimized with [Stochastic Gradient Descent (SGD)](https://en.wikipedia.org/wiki/Stochastic_gradient_descent). However, the loss function above only uses positive examples of interactions (e.g. clicks) which may produce *Folding:*



To avoid this phenomenon, we also use negative examples in the loss function, where the dot product between U\_i and I\_j should be near zero for items for which a user would unlikely click or positively interact. Thus, we augment the loss function above as:

Loss = \sum (A\_ij - U\_i \* I\_j)^2 + \lambda \sum (U\_ii \* I\_jj)^2

where the second term sums over negative samples and can be seen as a regularizing term. Also note that frequent users or items may dominate in the loss function; hence, we may need to weigh down the first term of the loss by the frequency of the user interactions or interactions that an item may have received in the past. Another common regularizing term is the L2 norm over the user embeddings and item embeddings:



In Collaborative Filtering, we may suffer the cold-start problem when an item or a user only has a few interactions. For example, if an item v has never appeared in the training data but the system has a few user interactions with this item, we can still estimate its embedding by doing:

argmin\_{v} || A\_.j - U \* v ||

Where the matrix A\_.j is the full column of A for item j, U is the user matrix which is kept constant, and v is the item embedding to be estimated. This estimation is possible without complete retraining, which is advantageous. Similarly we can do for a user that never appeared in the training data but for which we have some item interactions.

It might also be the case where a user or an item has absolutely no previous interactions; in this case, we can take the average of embeddings from items of the same category or users from the same region. We could also do clustering or K-nearest neighbors with hand-crafted features to find the most similar items or users and take the average across them.

Pros:

* We don’t need much knowledge about the items (e.g. news articles), since these recommendations are only based on other user’s interactions. The user and item embeddings are automatically learned.
* Serendipity: the system may help users to discover articles they are interested in but they didn’t know how to search, because it uses global information from all users.

Cons:

* This method suffers from the so-called cold-start problem: If there is a new movie, no-one else would’ve yet liked or watched it, so you’re probably not going to have this in your list of recommended movies, even if you’d love it.
* It is hard to explain to a user why something was recommended to them.
* Hard to include side features. We can solve by using a block matrix so that we also learn embeddings of features with matrix factorization.

### Retrieval

We have described functions to do light-weight scoring of items given recent items (content-based filtering) or similar users (collaborative filtering). Using these functions, how can we efficiently scan our vast stores of content to find the highest scoring items in practice? How can we efficiently exclude those undesirable items in a scalable manner?

**Fast retrieval of items**. Given a user or item representation, we can use K-nearest neighbors to find the K most similar items. However, there might be millions or billions of these items: it is very expensive to calculate similarities with each of these possible items. There are three typical similarity functions:

1. Dot product: sensitive to popularity and rare item high norms. In these cases, our query vector or item vector may have a large norm (because it is a popular item or because it was initialized like that), which would tend to produce larger dot product results and be systematically recommended more than other items.
2. Cosine similarity: only looks at the angle, independently on their vector lengths. This is robust against very popular items or rare items that got high norm during initialization.
3. We can also use approximate nearest neighbors, e.g. using [random projections](https://en.wikipedia.org/wiki/Random_projection) of the embedding/features or [LHS (locality-sensitive hashing)](https://en.wikipedia.org/wiki/Locality-sensitive_hashing).

These K-nearest neighbors are typically calculated offline, specially for *related items* recommendations since we only have to compute them once for each target item. For users, their user representation may change over time and K-NN has to be recomputed again, but that is possible at a certain cadence.

**Fast exclusion of items**. There are two types of items that we want to exclude from the recommended items: those that have already been visited, and those that have been flagged as *not desired* or similar to those. This exclusion can be implemented efficiently and in a scalable manner using [Bloom filter](https://llimllib.github.io/bloomfilter-tutorial/)s ([wikipedia](https://en.wikipedia.org/wiki/Bloom_filter)). These filters are *k* hash functions over *m* bits. Each item is hashed by all *k* hash functions and the corresponding bit of each hash function is set to 1. For any new item, we hash it using the *k* hash functions: if all its bits are set to 1, we say that *maybe* the item was already visited. If at least one bit is set to 0, we know that definitely this item was not visited.

Finally, we can exclude from the candidate list those news articles that are too similar to each other: displaying news articles with high redundancy makes us miss the opportunity to show other content that the user might be interested in and increase user satisfaction. [We can do this in a greedy manner](http://gdac.uqam.ca/WWW2016-Proceedings/companion/p87.pdf), starting from original content from popular media outlets and checking if other articles have a high cosine similarity to those from popular media.

### Moderation

We need to manage problematic content using the “[remove, reduce, and inform](https://about.fb.com/2018/05/inside-feed-reduce-remove-inform/)” strategy. This strategy involves *removing* content that violates our [Community Standards](https://www.facebook.com/communitystandards/), *reducing* the spread of problematic content that does not violate our standards, and *informing* people with additional information so they can choose what to click, read or share.

**Remove**. We will process every piece of content contributed by any entity such as a user, a group, media outlets, etc, and assign a confidence score to how likely this piece of content violates our Community Standards. If our confidence is above a certain threshold such that our true positive rate is above 99.9%, then an automatic enforcement action takes place to remove this content. However, this system will likely have a low recall. For pieces of content with a lower confidence score but above a certain threshold, they will be sent for manual moderation. Moreover, we can also add a button in the User Interface for users to report content that can be manually evaluated by our pool of knowledge associates. The objective of the system will be to reduce the amount of manual moderation and maximize the automated decisions.

* Authenticity: We can create featurizers, content similarity functions and hash collisions (e.g. Bloom Filters) to detect plagiarism and spam.
* Safety: We need to create classifiers to detect insults, offending symbols, threats, adult content and criminal content. As training data, we use previously manually moderated content and we extend using manually vetted external sources that are known to contain such offending material. We need to do this with text, images, text on images, video and audio.

**Reduce**. The reduce stage applies to pages, groups, profiles and friend recommendations, News Feed and its ranking. Through our Recommendations Guidelines, we work to avoid making recommendations that could be low-quality, objectionable, or particularly sensitive, and we also avoid making recommendations that may be inappropriate for younger viewers. Our Recommendations Guidelines are designed to maintain a higher standard than our Community Standards, because recommended content and connections are from accounts or entities you haven't chosen to follow. Therefore, not all content allowed on our platform will be eligible for recommendation. [Content not allowed for recommendations (but still allowed to exist in the platform):](https://www.facebook.com/help/1257205004624246)

1. Content that impedes our ability to foster a safe community, such as:
2. Sensitive or Low-Quality Content about Health or Finance
3. Content that Users Broadly Tell us they Dislike
4. Content that is associated with low-quality publishing,
5. False or Misleading Content

For this purpose, we need to create classifiers to detect these cases using historical data in each region. With the use of cross-lingual embeddings, we might be able to re-use models trained in one country (e.g the US) and be fine-tuned in a different country. For example, click baits are usually disappointing and recommending these to users may result in poor user experiences that we want to avoid. These click baits can be recognized automatically by using linguistic patterns from the title (e.g. “you are not going to believe [...]”, “three things that will surprise you [...]”).

**Inform**. Some pieces of news or blog posts may spread false or misleading information that can cause terrible outcomes. We could use external sites such as PolitiFact to label those that have been fact checked. We could also have semi-automated mechanisms to fact-check the top, most impactful pieces of news and to label or rate them in terms of how faithful they are to reality.

## [Scoring and Ranking](https://developers.google.com/machine-learning/recommendation/dnn/scoring)

The Candidate Generation stage has reduced the space of possible items to recommend from millions to just a few hundred. For a given user, now it is possible to use more sophisticated deep learning models to produce a score for each item; when we have scores for each item, we can simply rank them in descending order of score to produce the ranked item list.

### Features

Our features are either related to the query (user, history and context) or the item (textual news, photos, blog posts, groups, pages, potential friends). For continuous features, we need to normalize them by subtracting the average and dividing by the standard deviation that we observe on the training data. For categorical features, we can do a one-hot encoding and treat each category level as a separate feature.

Examples of user features are: gender, age, location, stated interests, marital status, status message, an aggregate of user embeddings from his/her top 3 friends, language, country of origin.

Examples of context features are: a sequence with the last 5-20 pieces of content the user clicked on (or spent more than 2 seconds on), the centroids of 5 clusters over the last one month of content interactions, the current day of the week, month of the year, time of the day, the top 5 local trending stories and top 5 national trending stories.

Examples of item features are: bag of word (BoW) representations of a news article (body or title), BoW of title from a photo shared by a friend, image encoding, distribution of objects detected in a posted image, concatenation of noun-phrase embeddings of top 10 TF-IDF terms, image encoding of a sample of video frames from different scenes, hashed ID of content producer, location of content producer, hashtags and other user mentions.

### Inference

At the inference stage, we would encode the user and context information into a single query vector of *d* dimensions. For each item generated at the candidate generation stage, we would use a different encoder depending on the modality of the item: we would have a specialized encoder for news articles, and a different encoder for posted photos from friends. Then, we would calculate the dot product between the query vector and each item vector to obtain a score that we hope to correlate with some target scores.

These target scores need to be proxies of our success business metrics such as click rate, watch time or session watch time. Our objective function will be to minimize the squared difference between our predicted score and one of these:

1. The click through rate (CTR). However, we will tend to recommend click-bait news and make the user lose interest in the long term.
2. The watch time. However, we will tend to recommend very long news articles, blog posts or videos but produce poor user experience because they are not so engaging.
3. Session time. This would be ideal since we would be recommending a variety of shorter pieces of content that are more engaging and the user will end up spending more time overall in the session (e.g. reading one piece of content after another in the same session). For this purpose, we would need to predict whether each item is going to be part of a session for which the overall session time is much larger than the item consumption time (e.g. a short video of 5 minutes in a facebook session of 2 hours).

Each item predicted score will hopefully be correlated with our target score and then we will sort these items in decreasing score so that items appearing on top of the user’s viewport will be the ones that maximize session time and engagement.

### Training

We can build our training data using our historical logs of user interactions. There will be two types of data:

1. User interactions (clicks, video watches, likes, etc.) on data that has not been recommended and hence will not be biased by the current recommendation engine. This is data the user has actively searched or accessed from other internal (friends’ wall) or external sources that Facebook has visibility into.
2. User interactions on recommended content. These interactions (e.g. *likes*) will be biased towards those items that appeared in the recommended list of the current recommender system. For example, if an item was not initially recommended, then it will not accumulate positive feedback from users and we will not collect enough training data to recommend similar items. To mitigate this, we can do an epsilon-greedy exploration for a small epsilon, to discover content patterns that are of interest to our users but that are not currently captured by our current recommender system.

We will also need to normalize the interactions that we want to predict. For example, if we want to recommend content that users *like* (e.g. push the button “like”), we would need to predict the *number of likes per 1000 views* so that items’ *likes* rate is comparable across items:

* Photo 1 received 100 views and 10 likes. It’s like rate is 0.1
* Photo 2 received 20 views and 10 likes. It’s like rate is 0.5, which is higher. Hence, we can say that Photo 2 is more engaging and likable, and we should learn the model parameters so that Photo 2 tends to be more recommended than Photo 1, even if both have the same number of *likes*.

In order to increase generalization and prevent *folding*, we need to obtain negative examples of items. These negative examples are items that should **not** be recommended to our users because these items are systematically ignored or disliked. Some sources of negative examples are:

1. News posts that have been reported, disliked or marked as *don’t show me more like this*. However, we are likely to have a very small percentage of these items: we need more negative examples.
2. News posts that were recommended but were not clicked nor liked by most users. This source is likely going to give us a larger scale dataset of negative examples; however, these examples are biased towards the recommendations that the current system produces and will not be representative of other negative examples outside of what our current recommender system is producing. Using this source, we can either do a random uniform sampling of these negative items or systematically select those for which the initial model (either product recommender system or our new experimental model in a preliminary stage) gave the highest scores. The latter is called *hard negatives* and they will contribute the most to the gradients and hence allow us to converge faster.
3. Negative examples obtained with an epsilon-greedy exploration where we show items that are selected using other methods such as TF-IDF in search or category-based in similar-items recommendations. These negative examples will not be biased.

But, what should be the right proportion in the training data of positive and negative examples? Regarding the positive examples, we could consider all examples that have at least X interactions (e.g. a minimum of 0.01 interactions per view). Regarding the negative examples, we could produce as many negative examples as positive ones, or we could introduce twice more negative examples than positive ones and then reduce to half the weight these negative examples have in the loss function. Using the latter method, we will expose the ML system to more patterns of errors and still keep the loss function balanced.

**Loss functions**. There are different loss functions that we could use depending on the target score to predict and the negative sampling scheme. If we want to predict whether an item is going to be *liked* or not (binary), then we could use the [binary cross-entropy loss](https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html):

−(𝑦 \* log(𝑝)+(1−𝑦)\* log(1−𝑝))

where *y* is the true label (e.g. the item has been clicked in the training data), and *p* is the predicted probability of being clicked. For example, for an item whose training label is y = 1 (*liked*), our system may produce a prediction of being liked of 0.7. Then, the loss would be

-(1 \* log(0.7) + 0 \* log(0.3)) = - log(0.7) = 0.35

However, if our system had predicted the probability to be 0.2 (worse prediction), the loss would be:

-(1 \* log(0.2) + 0 \* log(0.8)) = - log(0.2) = 1.6

which is a larger loss and for which larger gradients will be calculated and propagated to adjust the model parameters.

If we try to predict a continuous score, e.g. the ratio likes per 1000 views or the click-through rate (CTR), then we would use the [Mean Squared Error (MSE)](https://en.wikipedia.org/wiki/Mean_squared_error) between the gold score and our predicted score. For example, if we have an item in the training data for which the CTR is 0.3 and our predicted CTR is 0.1, the MSE will be (0.3 - 0.1) ^ 2 = 0.04.

Using a 1:1 ratio of positive to negative samples, one loss function could be:

Loss = \sum\_{up,ip} (y - EncUser(up; ) \* EncItem(ip, )) ^ 2 + \sum\_{un,in} (EncUser(un) \* EncItem(in)) ^ 2 + +

where up, ip are positive user and item pairs; un, in are negative user and item pairs, EncUser is the encoder of the user and the context, EncItem is the encoder of the item; are the parameters of the user encoder; are the parameters of the item encoder. With this loss function, we minimize the difference between the true user-item interaction and the dot product between each pair of users and items, we try to bring down to zero the dot product between negative pairs of users and items, and we use the L2 regularizer to keep the parameters of the user and item encoders as small as possible to prevent overfitting.

Given a model, training data and a loss function, we can use different optimizers to adjust the model parameters to minimize the loss function. These are some examples:

* [Stochastic Gradient Descent (SGD)](https://en.wikipedia.org/wiki/Stochastic_gradient_descent). This is a stochastic approximation to Gradient Descent which calculates gradients over the entire dataset. Instead, it randomly samples a small subset (stochastic mini-batch) and calculates the gradient on this mini-batch. This gradient is used to adjust the model parameters; this process is repeated iteratively over the whole training set, shuffling the examples each time to avoid memorizing the order of the examples. SGD has a lower convergence rate than GD, but it does faster iterations.
* [Momentum](https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Momentum): When using momentum, the previous weight update is recorded and the weight update at each iteration is a linear combination between the current weights, the negative gradient of the loss in the current observation, and the previous weight update multiplied by an exponentially decayed term , as follows:  . This makes the optimization more robust against oscillations and accelerates the learning when we are in parameter space that clearly needs acceleration.
* AdaGrad (Adaptive Gradient algorithm): It uses a different learning rate for each parameter.
* RMSProp.
* Adam.

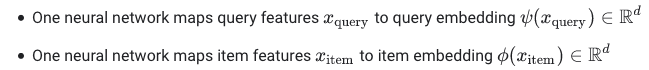
However, these optimizers do not know when to stop iterating. Iterating until convergence often produces a negative effect on the model performance because we are very likely overfitting the model to the training data. To avoid overfitting, it is common to:

* Use Dropout to introduce noise at some layers, specially fully connected layers.
* Use L1 and L2 regularizers as described above.
* Monitor the loss or accuracy on the validation set and stop iterating when the loss increases or the accuracy decreases consistently. This technique is often called Early Stopping and triggers after a certain number of iterations or epochs without improvements in the validation loss or accuracy. There is a *patience* parameter in Early Stopping that we can specify to wait a certain number of epochs until we stop the training process.

### Models

Deep Neural Networks can easily incorporate query features and item features due to the flexibility of the input layer of the network, which can help capture the specific interests and history of a user and improve the relevance of recommendations.

One possible architecture is a two-tower model:



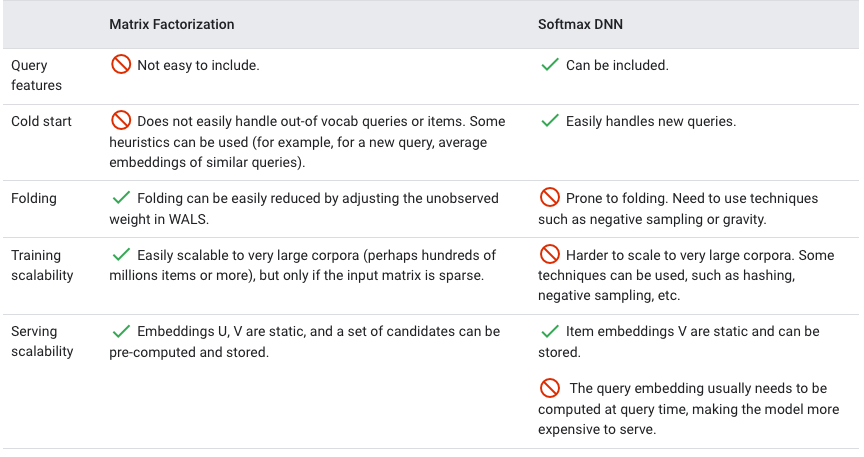
Then, these query and item embeddings are multiplied with each other using a dot product and we treat the result as the predicted relevance score. At the training stage, the parameters of these two networks should be adjusted so that this dot product correlates well with the score we want to predict (e.g. click-through rate, view time, session time or other forms of engagement).

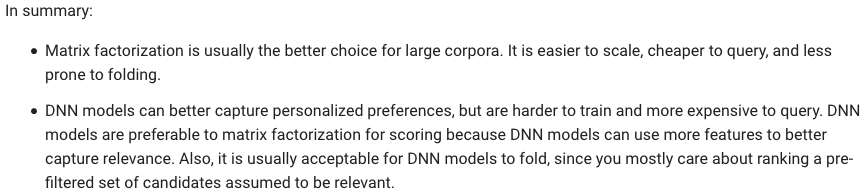
There are many possibilities in the creation of query embedding networks. We could start by having a separate Multilayer Perceptron to encode the user features (e.g. age, gender, marital status, country, etc.). Another branch would produce item encodings of the last 5 or 10 items, where each item encoder would be different depending on the modality. Then, the sequence of these item encodings will be processed by an LSTM and we would read-out the last hidden state of the LSTM and consider it as the history representation. This history representation will then be concatenated with the user embedding to form a single vector representing the user and the context. A simpler alternative to the LSTM is to use a weighted average of the history items, where recent items have a larger weight.

On the item embedding network, this will heavily depend on the modality of the item. For photos, we can have Convolutional Neural Networks (CNN) such as VGG16 to obtain a distribution over possible objects in the image. These networks are pre-trained on ImageNet to recognize a variety of objects from the ImageNet corpus which are photos linked to WordNet concepts. Alternatively, we could remove the top one or two layers in the VGG network and use intermediate feature representations as our image embeddings. We could either freeze the parameters of these pre-trained networks or fine-tune them to maximize the downstream recommendation performance. The use of these pre-trained networks and fine-tuning is known as transfer learning. For videos, we can sample a few representative frames from the video (in scenes that are visually very different), encode them as images and then concatenate them or do a MaxPooling on those vectors to obtain a single vector representation of the video. We could also concatenate image or video metadata to the embedding of the image or video.

News articles are mostly dominated by text in either the headline or the body. The encoding of this relatively large text can be done with unsupervised methods (e.g. doc2vec) or distant supervision using subcategories and cross-outlet article comparisons.

* [Doc2vec](https://datascience.stackexchange.com/questions/37488/doc2vec-how-does-the-inference-step-work-in-pv-dbow): this is an extension of word2vec, where document vectors are learned at training stage together with word vectors. At the inference stage, we need to do more learning steps to estimate the vector of the new input document. Since many news articles might be uploaded per minute, this procedure does not produce an acceptable latency and we will not consider it further.
* [BoW2Vec](http://gdac.uqam.ca/WWW2016-Proceedings/companion/p87.pdf): We can start by representing a document using the top 10,000 nouns that will form our vocabulary. Then, if a noun is present in the document, then we set that entry in the vector as 1 (we set to 0 otherwise). Then, we use a denoising auto-encoder to learn a mapping from the BoW to a hidden dimensional space of lower dimension (e.g. 300 dimensions). The auto-encoder should use two factors in the loss function: the reconstruction error and a triplet loss.
  + Loss = (prediction - y) ^ 2 + log(1 + exp(h1 \* h2 - h1 \* h3))
  + where *prediction* is the reconstructed document, *y* is the original document, the input *x* should have suffered some lossy transformation (e.g. masking/dropout noise at 0.3 rate), h1 is the hidden representation of the original document, h2 is the hidden representation of a *similar* document, and h3 is a hidden representation of a completely different document. Here, we intend to maximize the dot product between the hidden representations of the two documents that are similar.
  + The similar documents are obtained using distant supervision, e.g. news articles that are published at around the same time from different media outlets, using the same subcategories, keywords or named entities. Different documents can be easily obtained using documents from different months, or within the same media outlet and different news categories (e.g. sports vs. politics).





See [d2l model architectures for recommender systems](https://d2l.ai/chapter_recommender-systems/index.html) to possibly see other alternatives.

## Re-Ranking

* 1. re-rank the candidates to consider additional criteria or constraints:
     1. Constraints:
        1. Filter out items similar to the ones the user flagged as “show me less like this”.
        2. Filter out already-consumed items.
     2. Additional criteria:
        1. Freshness: incorporate the latest usage information and newest items.
           1. Re-train as often as possible with a warm start to reduce training time.
           2. Create an “average” user embedding to represent new users, possibly using clusters of users according to user profile features.
           3. Use DNNs since they accept arbitrary input features for users or items. This allows recommendations for newly created users and items.
           4. Add the document's age or last time since viewing as a feature and let the model learn weights for that feature.
        2. Diversity: avoid recommending too similar content to what the user just consumed because this is boring and disengaging.
           1. Train multiple candidate generators.
           2. Train with different objective functions.
           3. Re-rank based on genre or other metadata.
        3. Fairness: to treat users fairly and avoid learning unconscious biases from the training data.
           1. Include diverse perspectives in design and development.
           2. Train ML models on comprehensive data sets. Add auxiliary data when your data is too sparse (for example, when certain categories are under-represented).
           3. Track metrics (for example, accuracy and absolute error) on each demographic to watch for biases.
           4. Make separate models for underserved groups.

## [Metrics](https://en.wikipedia.org/wiki/Recommender_system#Performance_measures)

There are two types of metrics that we need to track at different stages of development: offline metrics and online metrics. Offline metrics are useful to do fast iterations in our model development, e.g. to experiment with different features, model architectures and hyperparameters. Online metrics are useful to understand the real business and user impact of our models, typically in an A/B testing experiment.

Offline and online metrics should be broken down per demographic group, specially on those protected statuses so that we ensure that the true positive rate (recall) is high enough for all populations: this is an essential component of a ***fair*** system.

### Offline metrics

* [Mean Reciprocal Rank (MRR)](https://en.wikipedia.org/wiki/Mean_reciprocal_rank): We may only be interested in generating recommendations for which only the first click is important because that would take the user to a different page or a different journey out of the News Feed. If that is the case, we can use MRR, which is defined as:



where |Q| is the number of queries, and rank\_i is the position of the clicked recommendation. If users click always on the first recommended item, the fraction will be 1/1 = 1 and MRR will be 1.

* Normalized Discounted Cumulative Gain (NDCG, [wikipedia](https://en.wikipedia.org/wiki/Discounted_cumulative_gain), [medium](https://towardsdatascience.com/evaluate-your-recommendation-engine-using-ndcg-759a851452d1)): The *normalized* Discounted Cumulative Gain is the Discounted Cumulative Gain divided by the ***Ideal*** Discounted Cumulative gain: 

where DCG is defined as:



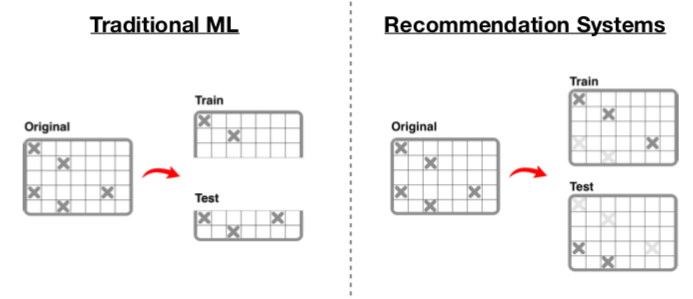
and IDCG is:



where p is the total positions in the recommendations list, *i* iterates over each position, and rel\_i is the relevance at each position, which is typically a number between 0 and 5. That is, highly relevant items in lower positions count more towards the score. In order to calculate nDCG, we need manually annotated relevances for queries which is an expensive manual process and we typically can only afford to have a few of them for evaluation or validation.

* [**Precision at K**](https://stackoverflow.com/questions/55748792/understanding-precisionk-apk-mapk)(P@K):How many relevant items are present in the top-k recommendations of your system. For example, to calculate P@3: take the top 3 recommendations for a given user and check how many of them are good ones. That number divided by 3 gives you the P@3. Note that Precision and Recall in this context are only defined for binary outcomes (relevant vs. not relevant). If we have relevance scores or labels (e.g. relevance {1, 2, 3, 4, 5}), then we would need to binarize them, e.g: every item with a relevance 3 or above should be considered relevant. The others would be irrelevant.
* Average Precision at K (AP@K): The mean of P@i for i=1, ..., K. For example, to calculate AP@3: sum P@1, P@2 and P@3 and divide that value by 3. AP@K is typically calculated for one user.
* Mean Average Precision at K (mAP@K): The mean of the AP@K for all the users. For example, to calculate MAP@3: sum AP@3 for all the users and divide that value by the amount of users.
* Recall at K (R@K): (# of recommended items @k that are relevant) / (total # of relevant items). For example, if our manually annotated dataset has 10 items that are relevant, and our recommender system only ranks 5 of them on the top 10 (the rest of items in the top 10 are either irrelevant or we don’t have a relevance score for them), then our Recall@10 will be 50%.
* [Kendall’s Tau](https://en.wikipedia.org/wiki/Kendall_tau_distance) if we have gold orderings. This is the proportion of item pairs that have the opposite relative order as in the gold data. That is, for each pair of items in the recommended list, we check whether their relative order is the same as in the gold data. For example, in predicted recommendations, we may have this order: A1, B1, C1. In the gold order, we have A2, C2, B2. The case where we have opposite orders is B1 < C1 compared to C2 < B2. There are 3 possible total orderings, so Kendall's tau distance is 1 / 3 = 0.333.
* [The area under the ROC curve (AUC)](https://en.wikipedia.org/wiki/Receiver_operating_characteristic): Is the integral (area below) the ROC curve and it is useful to compare two recommender systems regardless of the relevance threshold that they use. The ROC curve has the Recall (true positive rate) on the y-axis and the false alarm rate (false positive rate) on the x-axis, for all possible decision thresholds.
* Precision-Recall Curve: This is the curve where the Precision is plotted on the y-axis and the Recall on the x-axis, for every possible threshold. Two systems can be easily compared by overlaying their Precision-Recall curves.
  1. Another typical way to measure two systems independent on their thresholds is by fixing the Precision at 90% (or another percentage guided by the business) and then comparing their Recalls. The system with the highest recall at P@90% would be the best system.
* Root Mean Square Error (RMSE): It calculates the square error between the predicted rating and the true (historical) rating, then calculates the mean and the square root of it.

Note that in traditional machine learning, we split our original dataset to create a training set and a validation set. This, however, doesn’t work for recommender models since the model won’t work if we train all of our data on a separate user population and validate it on another. Instead, mask some entries in the ratings matrix, train and predict on them:



### Online metrics

At different points of time, we may want to optimize more one metric than others. E.g. for a new user, we may want to maximize satisfaction, then later transit to discovery of new artists.

* 1. CTR (#clicks/#imps), depth of scroll (#imps/#sessions)
  2. **User Stickiness** is generally calculated as the ratio of Daily Active **Users** to Monthly Active **Users**. A DAU/MAU ratio of 50% would mean that the average **user** of your app is using it 15 out of 30 days that month
* Training and loss function
  1. Automated data quality checks, Data balancing, Monitoring, early stopping
* A/B testing and online evaluation
  1. Hypothesis testing, p-values
  2. [P-value](https://en.wikipedia.org/wiki/P-value): the ***p*-value**[[note 1]](https://en.wikipedia.org/wiki/P-value#cite_note-2) is the probability of obtaining test results at least as extreme as the [results actually observed](https://en.wikipedia.org/wiki/Realization_(probability)), under the assumption that the [null hypothesis](https://en.wikipedia.org/wiki/Null_hypothesis) is correct. A very small *p*-value means that such an extreme observed [outcome](https://en.wikipedia.org/wiki/Outcome_(probability)) would be very unlikely under the null hypothesis. For example, if we do an A/B test with a new recommender system and we see an increase of average revenue with a p-value of 0.01, we know that there is only a 1% probability that the increase in revenue is purely random: that is, our new recommender system is likely causing the increase of revenue.
  3. Dialing up and down
* User feedback, retraining and adaptation
  1. Feedback:
     1. When a user clicks on a piece of news or forwards it. We can record the time of the day to calculate news type propensity over the day. This is online content-based filtering.
     2. Interactions with friends or groups. We can recommend news similar to those consumed by the friends or aggregation in the group. This is online collaborative filtering.
     3. Explicit vs. implicit. Explicit: rating, “like”, “heart”, “save”, “share”, “show me less like this”, “hide”, “report”. Implicit: clicks, time spent, impression time.
     4. Study how feedback could be used in each of the modules: candidate generation (both content-based and collaborative filtering), ranking and re-ranking.
* Monitoring
  1. Latency
  2. Changes in output probability distribution and input distributions
  3. Success metrics

## Others

Data:

* 1. we need to think about how an algorithm will handle three kinds of inputs: models, data, and signals.
     1. Models are usually small files of parameters that have been previously trained offline.
     2. Data is previously processed information that has been stored in some sort of database, such as movie metadata or popularity.
     3. We use the term “signals” to refer to fresh information we input to algorithms. This data is obtained from live services and can be made of user-related information, such as what the member has watched recently, or context data such as session, device, date, or time.
* Deployment
  1. Batch processing
  2. Real time processing (serving)
  3. Replication
  4. Load balancers
* Personal data protection and privacy compliance
* *Equal opportunity*, introduced in [2016\_NIPS Equality of Opportunity in Supervised Learning](https://dl.acm.org/citation.cfm?id=3157469), is the concept that the *true positive rate* should be the same across groups. We would need to break down our true positive rate across demographic variables and observe differences.
  1. More on [Fairness](http://www.ec.tuwien.ac.at/~dimitris/research/recsys-fairness.html).