Tweet Selection

Let's see how the Tweet selection component fetches Tweets to display on a user's Twitter feed.

We'll cover the following

- Tweet selection schemes
  - New Tweets
  - New Tweets + unseen Tweets
    - Edge case: User returning after a while
  - Network Tweets + interest / popularity-based Tweets

Tweet selection schemes

Let's see the various schemes Tweet selection component uses to fetch Tweets for a user's twitter feed.

New Tweets

Consider the following scenario to see how Tweet selection occurs.

User *A* logs in to Twitter at 9 am to view their Twitter feed. Now, the Tweet selection component has to fetch Tweets for display. It fetches the five-hundred newly generated Tweets by A's network since the last login at 3 pm yesterday.

Select new Tweets generated by A's network between 3pm to 9 am

> *A not-so-new Tweet!*
>
> Consider a Tweet that user A has viewed previously. However, by the time the user logs in again, this Tweet has received a much bigger engagement and/or A's network has interacted with it. In this case, it once again becomes relevant to the user due to its recent engagements. Now, even though this Tweet isn't new, the Tweet selection component will select it again so that the user can view the recent engagements on this Tweet.

New Tweets + unseen Tweets

Picking up where we left off, let's see how this component may select a mix of new and old unseen Tweets.

The new Tweets fetched in the previous step are ranked and displayed on user A's feed. They only view the first two-hundred Tweets and log out. The ranked results are also stored in a cache.

Now, user A logs in again at 10 pm. According to the last scheme, the Tweet selection component should select all the new Tweets generated between 9 am and 10 pm. However, this may not be the best idea!

For 9am's feed, the component previously selected a Tweet made at 8:45 am. Since this Tweet was recently posted, it did not have much user engagement at that time. Therefore, it was ranked at the 450th position in the feed. Now, remember that A logged out after only viewing the first two-hundred Tweets and this Tweet remained unread. Since the user's last visit, this unread Tweet gathered a lot of attention in the form of reshares, likes, and comments. The Tweet selection component should now reconsider this Tweet's selection. This time it would be ranked higher, and A will probably view it.

Keeping the above rationale in mind, the Tweet selection component now fetches a mix of newly generated Tweets along with a portion of unseen Tweets from the cache.

Select a mix of newly generated Tweets and unseen Tweets from the cache

You may encounter the following edge case during Tweet selection.

Edge case: User returning after a while

Consider a scenario, where user *A* might log in after two weeks. A lot of Tweets would have been generated since A's last login. Therefore, the Tweet selection component will impose a limit on the Tweet data it will select. Let's assume that the limit is five-hundred. Now the selected five-hundred Tweets may have been generated over the last two days or throughout the two weeks, depending on the user's network's activity. The main focus is that the pool of Tweets keeps on increasing so a limit needs to be imposed on the number of Tweets that the component will fetch.

Limit imposed on the number of tweets selected

Network Tweets + interest / popularity-based Tweets

Until now, the Tweet selection component was only fetching Tweets from user A's network of followees. However, there could be Tweets **outside of user A's network** that have a high potential of engaging them. Hence, we arrive at a two-dimensional scheme of selecting network Tweets and potentially engaging Tweets.

An engaging Tweet could be one that:

- aligns with user A's interests
- is locally/globally trending
- engages user A's network

Selects network Tweets and interest / popularity-based Tweets

Selecting these Tweets can prove to be very beneficial in two cases:

1. The user has recently joined the platform and follows only a few others. As such, their small network is not able to generate a sufficient number of Tweets for the Tweet selection component (known as the Bootstrap problem).
2. The user likes a Tweet from a person outside of his network and decides to add them to their network. This would increase the discoverability on the platform and help grow the user's network.

> All of the schemes we have discussed showcase the dynamic nature of the Tweet selection component's task. The Tweets it will select will continue to change each time.

Feature Engineering

Let's engineer some features for the Tweet ranking model.

We'll cover the following

- Features for the ranked
- Dense features
- User-author features
  - User-author historical interactions
  - User-author similarity
- Author features
  - Author's degree of influence
  - Author's historical interactions on the author's Tweets
- User Tweet features
  - Tweet features
  - Tweet features on the Tweet's content
  - Interaction based on Tweet's interaction
  - Separate features for different engagements
- Content-based features
  - Sparse features

The machine learning model is required to predict user engagement on user-Tweet both (ranked engagement) because it is the main ranked engagement.

The features are shown here is the result of our brainstorm session. However, feature engineering is an iterative process. As an iterative, we need to keep on brainstorming to think about more features.

Let's begin by identifying the four main **actors** in a twitter feed:

1. The logged-in user
2. The author
3. The Tweet
4. The context

Logged-in User
Author
Tweet
Context
Age
Gender
Location
Interests
Following
Degree
Reputation
Topics
Tweet Age
Tweet Type
Tweet Length
Hashtags
Mentions
Media
Time of Day
Day of Week
Device
Location
Author's Tweets
Author's Interactions
Tweet's Interactions
Tweet's Content
Author's Degree
Author's Reputation
Author's Topics
Author's Tweet Type
Author's Tweet Length
Author's Hashtags
Author's Mentions
Author's Media
Author's Location
Author's Interests
Author's Following
Author's Age
Author's Gender
Author's Location
Tweet's Interactions
Tweet's Content
Tweet's Hashtags
Tweet's Mentions
Tweet's Media
Tweet's Location
Tweet's Interests
Tweet's Age
Tweet's Type
Tweet's Length
Tweet's Time
Tweet's Day
Tweet's Device
Tweet's Location
Author's Tweets
Author's Interactions
Author's Degree
Author's Reputation
Author's Topics
Author's Tweet Type
Author's Tweet Length
Author's Hashtags
Author's Mentions
Author's Media
Author's Location
Author's Interests
Author's Following
Author's Age
Author's Gender
Author's Location
Author's Tweets
Author's Interactions
Author's Degree
Author's Reputation
Author's Topics
Author's Tweet Type
Author's Tweet Length
Author's Hashtags
Author's Mentions
Author's Media
Author's Location
Author's Interests
Author's Following
Author's Age
Author's Gender
Author's Location
Tweet's Interactions
Tweet's Content
Tweet's Hashtags
Tweet's Mentions
Tweet's Media
Tweet's Location
Tweet's Interests
Tweet's Age
Tweet's Type
Tweet's Length
Tweet's Time
Tweet's Day
Tweet's Device
Tweet

Training Data Generation

Let's collect and label training data for the feed ranking ML model.

We'll cover the following

- Training data generation through online user engagement
- Balancing positive and negative training examples
- Train test split

Your user engagement prediction model's performance will depend largely on the quality and quantity of the training data. So, let's see how you can generate training data for your model.

> Note that the term *training data row* and *training example* will be used interchangeably.

Training data generation through online user engagement

The users' online engagement with Tweets can give us positive and negative training examples. For instance, if you are training a single model to predict user engagement, then all the Tweets that received user engagement would be labeled as positive training examples. Similarly, the Tweets that only have impressions would be labeled as negative training examples.

> **Impression**: If a Tweet is displayed on a user's Twitter feed, it counts as an impression. It is not necessary that the user reads it or engages with it, scrolling past it also counts as an impression.

Training data generation for a single model to predict user engagement

Any user engagement counts as positive example

However, as you saw in the architectural components lesson, that you can train different models, each to predict the probability of occurrence of different user actions on a tweet. The following illustration shows how the same user engagement (as above) can be used to generate training data for separate engagement prediction models.

Training data generation for separate models to predict user engagement

When you generate data for the "Like" prediction model, all Tweets that the user has liked would be positive examples, and all the Tweets that they did not like would be negative examples.

> Note how the comment is still a negative example for the "Like" prediction model.

Similarly, for the "Comment" prediction model, all Tweets that the user commented on would be positive examples, and all the ones they did not comment on would be negative examples.

Balancing positive and negative training examples

Models essentially learn behavior from the data we present them with. Therefore, it's important for us to provide a good sample of both positive and negative examples to model these interactions between different actors in a system. In the feed-based system scenario, on average, a user engages with as little as approximately 5% of the Tweets that they view per day. How would this percentage affect the ratio of positive and negative examples on a larger scale? How can this be balanced?

Looking at the bigger picture, assume that one-hundred million Tweets are viewed collectively by the users in a day. With the 5% engagement ratio, you would be able to generate five million positive examples, whereas the remaining ninety-five million would be negative.

Let's assume that you want to limit your training samples to ten million, given that the training time and cost increases as training data volume increases. If you don't do any balancing of training data and just randomly select ten million training samples, your models will see only 0.5 million positive examples and 9.5 million negative examples. In this scenario, the models might not be able to learn *key* positive interactions.

Therefore, in order to balance the ratio of positive and negative training samples, you can **randomly downsample**:

- negative examples to five million samples
- positive examples to five million samples

Now, you would have a total of **ten million training examples per day**; five million of which are positive and five million are negative.

> **Note**
> If a model is *well-calibrated*, the distribution of its predicted probability is similar to the distribution of probability observed in the training data. However, as we have changed the sampling of training data, our model output scores will **not** be well-calibrated. For example, for a tweet that got only 5% engagement, the model might predict 50% engagement. This would happen because the model is trained on data that has an equal quantity of negative and positive samples. So the model would think that it is as likely for a tweet to get engagement as it is to be ignored. However, we know from the training that this is not the case.
> Given that the model's scores are only going to be used to rank Tweets among themselves, poor model calibration doesn't matter much in this scenario. We will discuss this in the ads system chapter, where calibrated scores are important, and we need to be mindful of such a sampling technique.

Train test split

You need to be mindful of the fact that the user engagement patterns may differ throughout the week. Hence, you will use a week's engagement to capture all the patterns during training data generation. At this rate, you would end up with around seventy million rows of training data.

You may randomly select $\frac{2}{3}^{rd}$, or 66.6%, of the seventy million training data rows that you have generated and utilize them for training purposes. The rest of the $\frac{1}{3}^{rd}$, or 33.3%, can be used for validation and testing of the model.

However, this random splitting defeats the purpose of training the model on an entire week's data. Also, the data has a time dimension, i.e., we know the engagement on previous Tweets, and we want to predict the engagement on future Tweets ahead of time. Therefore, you will train the model on data from one time interval and validate it on the data with the succeeding time interval. This will give a more accurate picture of how the model will perform in a real scenario.

> We are building models with the intent to forecast the future.

In the following illustration, we are training the model using data generated from the first and second week of July and data generated in the thirdweek of July for validation and testing purposes.

Splitting data for training, validation, and testing

> Have a look at the training data collection strategies lesson to get more insights.

Ranking

Now, we'll explore different modeling options for the Tweet ranking problem.

We'll cover the following

- Modeling options
  - Logistic regression
  - MART
  - Deep learning
    - Separate neural networks
    - Multi-task neural networks
  - Stacking models and online learning

Modeling options

Some questions to consider are: Which model will work best for this task? How should you set up these models? Should you use one globally different models for each task? Let's go over the modeling options and try to answer these questions. We will also discuss the pros and cons of every approach.

Logistic regression

Initially a simple model that makes sense to train is a logistic regression model with regularization, to predict engagement using the dense features that we discussed in the feature engineering lesson.

A key advantage of using logistic regression is that it is reasonably fast to train. This enables you to test new features fairly quickly so and they make an impact on the ACC or calibration error. Also, it's extremely easy to understand the model. You can see from the feature weights which features have turned out to be more important than others.

A major limitation of the linear model is that it assumes linearity exists between the input features and prediction. Therefore, you have to manually model feature interactions. For example, if you believe that the age of the user(feature a) major holiday(feature b) have a strong impact on your engagement prediction, you will have to create this feature in your training data manually. Other models like tree-based and neural networks are able to learn these feature interactions and utilize them effectively for prediction.

Another key question is whether you want to train a single classifier for overall engagement or separate ones for each engagement action based on production needs. In a single classifier case, you can train a logistic regression model for predicting the overall engagement on a Tweet. Tweets with any form of engagement will be considered as positive examples for this approach.

Another approach is to train separate logistic regression models to predict Pclick, Pcommentor and Preweet.

These models will utilize the same features. However, the assignment of labels to the training examples will differ and, hence, the weights, intercepts, or removed will be considered positive examples for the overall engagement predictor model. Only Tweets with likes will be considered as positive examples for the "Liker" predictor model. Similarly Tweets with comments will be considered as positive examples for the "Commentor" predictor model and only Tweets with retweets will be considered as positive examples for the "Retweet" predictor model.

If As with any modeling, we should try and tune all the hyperparameters and play with different features to find the best model to fit this problem.

MART

If MART stands for multiple additive regression trees.

Another modeling option that should be able to outperform logistic regression with dense features is additive tree-based models, e.g. Boosted Decision Trees and Random Forests. Tree-based models are able to learn feature interactions between the dense features that are mostly available in logistic regression as discussed in the above section.

Tree-based models also don't require a large amount of data as they are able to generalize well quickly. So, a few million examples should be enough for a reasonable model.

There are various hyperparameters that you might want to play around to get an optimized model, including:

- Number of trees
- Maximum depth
- Minimum samples needed for split
- Maximum features sampled for a split

To choose the best predictor, you should do hyperparameter tuning and select the hyperparameter that minimizes the cost error.

Approach 1

A simple approach is to train a single model to predict the overall engagement.

Approach 2

You could have specialized predictors to predict different kinds of engagement. For example, you can have a like predictor, a retweet predictor and a reshare predictor. Once again these models will utilize the same features. However, the assignment of labels to the training examples will differ depending on the model's objective.

Approach 3

Consider a scenario, where a person reshares a Tweet but does not click the like button. Even though the user didn't actually click on the button, retweeting generally implies that the user like the Tweet. The positive training example for the retweet model may prove useful for the like model as well. Hence, you can reuse all positive training examples across every model.

One way to utilize the overall engagement data among each individual predictor of Pclick, Pcomment and Preweet is to build one common predictor, i.e., Pengagement and share its output as input into all of your predictors.

If To take approach three a notch further, you can use the output of each tree in the "overall engagement predictor" ensemble as features in the individual predictors. This allows for even better learning as the individual model will be able to learn from the output of each individual tree.

Deep learning

With advancements in computational power and the amount of training data you can gather from hundreds of million users, deep learning can be very powerful in predicting user engagement for a given Tweet and Ranking can see a lightly personalized feed.

Having said that, training the model as well as evaluating the model at great generation time can make this approach quite expensive. So, you may want to fall back to the multi-layer approach, which we discussed in the search ranking chapter, i.e., having a simple model for stage one ranking and use complex single two-tower model to score the most relevant Tweets ranked at the top of the user's Twitter feed.

Like with the tree-based approach, there are quite a few hyperparameters that you should tune for deep learning to find the most optimized model that minimizes the test set error. They are:

- Learning rate
- Number of hidden layers
- Batch size
- Number of epochs
- Dropout rate for regularizing model and avoiding overfitting

If Neural networks are a great choice to add all the raw data and dense features that we used for tree-based models.

Separate neural networks

One way is to train separate neural nets for each of the Pclick, Pcomment and Preweet.

However, for a very large scale data set, training separate deep neural networks (NNs) can be slow and take a very long time to train and hence to deploy. The following approach of setting up the models for multi-task learning caters to this problem.

Multi-task neural networks

Likes, comments, and retweets are all different forms of engagement on a Tweet. Therefore, predicting Pclick, Pcomment and Preweet are similar tasks. When predicting similar tasks, you can use multi-task learning. Referring back to the example in approach 3 of MART, if you want to predict the probability of a like, it would certainly be helpful for the model to know that this Tweet has been retweeted (knowledge sharing). Also, you can train a model with shared layers and task specific fine-grained knowledge appended with the output of the shared layers. In addition, the weights of the shared layers are common to the three tasks. Whereas in the task-specific layer, the network learns information specific to the task. The loss function for this model is the sum of individual losses for all the tasks:

total_loss = like_loss + comment_loss + retweet_loss

The model will then optimize for this joint loss leading to regularization and joint learning.

This approach should be able to perform at least as effective as training separate networks for each task. It should be able to outperform in most cases as we share the shared data and use it across each predictor. Also, one key advantage of using shared layers is that models would be much faster to train than training completely separate deep neural networks for each task.

Stacking models and online learning

One way to superlative the "single model technique approach" is to use multiple models to utilize the power of different techniques by stacking models on top of each other. Let's go over one such stacking setup that should work out very well for the feed problem.

The setup includes training tree-based models and neural networks to generate features that you will utilize in logistic regression model. The main advantage of doing this is that you can utilize online learning, meaning we can continue to update the model with every user action on the live site. You can also use sparse features in our tree-based models giving the power of all non-linear relations and functions through feature generated by tree-based models and neural networks.

In the above diagram, each leaf node in the random forest will result in a boolean feature that will be active based on whether the leaf node is triggered or not.

Similarly, for neural networks, rather than predicting the probability of events, you can just plug in the output of the last hidden layer as features into the logistic regression model.

To summarize, this stacking model setup will give us all the learning power of deep neural networks and tree-based models along with the flexibility of training logistic regressions model, while keeping it almost real-time refreshed with online learning.

If Logistic regression can be easily used for changing the model online, based on real-time user interactions that happen with the tweets. This helps the model to re-learn the weights of all live features as well.

Another advantage of using tree-based models with logistic regression is that you can also utilize sparse features to learn the interactions, e.g., features like user_id and tweet_id can be used to memorize the interactions with each individual user and Tweet.

Given that features like tweet_id and user_id are extremely sparse, training and evaluation of the model must be done in a distributed environment because the data won't fit on one machine.

Diversity

Let's learn methods to reduce monotony on the user's Twitter feed.

We'll cover the following

- Why do you need diverse Tweets?
  - Diversity in Tweets' authors
  - Diversity in tweets' content
- Introducing the repetition penalty

Why do you need diverse Tweets?

Let's assume that you adopted the following modelling option for your Twitter feed ranking system. The Tweet selection component will select one-hundred thousand Tweets for user A's Twitter feed. The stage one ranker will choose the top five-hundred most engaging Tweets for user A. The stage two ranker will then focus on assigning engagement probabilities to these Tweets with a higher degree of accuracy. Finally, the Tweets will be sorted according to the engagement probability scores and will be ready for display on the user's Twitter feed.

The layered model approach

Consider a scenario where the sorted list of Tweets has five consecutive posts by the same author. No, your ranking model hasn't gone bonkers! It has rightfully placed these tweets at the top because:

- The logged-in user and the Tweet's author have frequently interacted with each other's Tweets
- The logged-in user and the Tweet's author have a lot in common like hashtags followed and common followees
- The author is very influential, and their Tweets generally gain a lot of traction

> 📝 This scenario remains the same for any modelling option.

Diversity in Tweets' authors

However, no matter how good of a friend the author is or how interesting their Tweets might be, user A would eventually get bored of seeing Tweets from the same author repeatedly. Hence, you need to introduce diversity with regards to the Tweets' author.

Diversity in tweets' content

Another scenario where we might need to introduce diversity is the Tweet's content. For instance, if your sorted list of Tweets has four consecutive tweets that have videos in them, the user might feel that their feed has too many videos.

Introducing the repetition penalty

To rid the Twitter feed from a monotonous and repetitive outlook, we will introduce a repetition penalty for repeated Tweet authors and media content in the Tweet.

One way to introduce a repetition penalty could be to add a negative weight to the Tweet's score upon repetition. For instance, in the following diagram, whenever you see the author being repeated, you add a negative weight of -0.1 to the Tweet's score.

Repetition penalty for repeated Tweet author

Another way to achieve the same effect is to bring the Tweet with repetition three steps down in the sorted list. For instance, in the following diagram, when you observe that two consecutive Tweets have media content in them, you bring the latter down by three steps.

Repetition penalty for consecutive Tweets with media content

Online Experimentation

Let's see how to evaluate the model's performance through online experimentation.

We'll cover the following

- Step 1: Training different models
- Step 2: Validating models offline
- Step 3: Online experimentation
- Step 4: To deploy or not to deploy

Let's look at the steps from training the model to deploying it.

Step 1: Training different models

Earlier, in the training data generation lesson, we discussed a method of splitting the training data for training and validation purposes. After the split, the training data is utilized to train, say, **fifteen** different models, each with a different combination of hyperparameters, features, and machine learning algorithms.

Different models are trained to predict user engagement

The above diagram shows different models that you can train for our tweet engagement prediction problem. Several combinations of feature sets, modeling options, and hyperparameters are tried.

Step 2: Validating models offline

Once these **fifteen** models have been trained, you will use the validation data to select the best model offline. The use of unseen validation data will serve as a sanity check for these models. It will allow us to see if these models can generalise well on unseen data.

Each model's performance is observed on the validation data

Step 3: Online experimentation

Now that you have selected the best model offline, you will use A/B testing to compare the performance of this model with the currently deployed model, which displays the feed in reverse chronological order. You will select 1% of the five-hundred million active users, i.e., five million users for the A/B test. Two buckets of these users will be created each having 2.5 million users. Bucket one users will be shown twitter timelines according to the time-based model; this will be the control group. Bucket two users will be shown the Twitter timeline according to the new ranking model.

Bucket one users see the control version, whereas Bucket two users see the varied version of the Twitter timeline

However, before you perform this A/B test, you need to retrain the ranking model.

Recall that you withheld the most recent partition of the training data to use for validation and testing. This was done to check if the model would be able to predict future engagements on tweets given the historical data. However, now that you have performed the validation and testing, you need to retrain the model using the recent partitions of training data so that it *captures the most recent phenomena*.

Step 4: To deploy or not to deploy

The results of the A/B tests will help decide whether you should deploy the new ranking model across the platform.

Engagement aggregates for both buckets of users

You can observe that the Twitter feeds generated by the new ranking model had thirty (180k-150k) more engagements.

Increase in engagement (*gain*) = $\frac{180{,}000 - 150{,}000}{150{,}000} * 100 = 20\%$

This model is clearly able to outperform the current production, or live state. You should use statistical significance (like p-value) to ensure that the gain is real.

> Learn more about online experimentation in this lesson.

Another aspect to consider when deciding to launch the model on production, especially for *smaller gains*, is the increase in complexity. If the new model increases the complexity of the system significantly without any significant gains, you should not deploy it.

> You should go for complex solutions (based on new features, or data, etc.) only if you anticipate it to bring larger gains in the future.

To wrap up, if, after an A/B experiment, you see an engagement gain by the model that is statistically significant and worth the complexity it adds to the system, it makes sense to replace the current live system with the new model.

Problem Statement

Let's get acquainted with the task of building a recommendation system.

We'll cover the following

- Introduction
- Problem statement
- Visualizing the problem
- Scope of the problem
- Problem formulation
  - Types of user feedback

Introduction

Recommendation systems are used by most of the platforms we use daily.

For example:

1. The Amazon homepage recommends personalized products that we might be interested in.
2. The Pinterest feed is full of pins that we might like based on trends and our historical browsing.
3. Netflix shows movie recommendations based on our taste, trending movies, etc.

In this chapter, we will go over the Netflix movie recommendation problem, but a similar technique can be applied to nearly all other recommendation systems.

Problem statement

The interviewer has asked you to **display media (movie/show) recommendations for a Netflix user.** Your task is to make recommendations in such a manner that the chance of the user watching them is maximized.

Visualizing the problem

The prime factor that led to Netflix's success was its recommendation system. The algorithm does a great job of bringing the right kind of content to the right users. Unlike Netflix's recommendation system, a simple recommendation system would have simply recommended the trending movies/shows with little regard for the particular user's preferences. At most, it would look at the viewer's past watches and recommend movies/shows of the same genre.

Another key aspect of Netflix's approach is that they have found ways to recommend content that seemed different from the user's regular choices. However, Netflix's recommendations are not based on wild guesses, but they are based on *other users' watch histories*, these users share some common patterns with the concerned user. This way, customers got to discover new content that they wouldn't have found otherwise.

> 80% of the shows watched on Netflix are driven by its recommendations, as opposed to someone searching for a particular show and watching it.

The task at hand is to create such a recommendation system that keeps the viewers hooked and introduces them to varied content that expands their horizons.

Scope of the problem

Now that you know the problem at hand, let's define the scope of the problem:

1. The total number of subscribers on the platform as of 2019 is 163.5 million.
2. There are 53 million international daily active users.

Hence, you have to build a system for a large number of users who require good recommendations on a daily basis.

One common way to set up the recommendation system in the machine learning world is to pose it as a classification problem with the aim to predict the probability of user engagement with the content. So, the problem statement would be:

"Given a user and context (time, location, and season), predict the probability of engagement for each movie and order movies using that score."

Problem formulation

We established that you will predict the probability of engagement for each movie/show and then order/rank movies based on that score. Moreover, since our main focus is on getting the users to watch most of the recommendations, the recommendation system would be based on implicit feedback (having binary values: the user has watched movie/show or not watched).

Let's see why we have opted for ranking movies using "implicit feedback" as our probability predictor instead of using "explicit feedback" to predict movie ratings.

Minimize

Build a recommender system that predicts the user rating for the media. The goal would be to show such recommendations where users give high ratings.

Types of user feedback

Generally, there are two types of feedback coming from an end-user for a given recommendation.

1. **Explicit feedback:**

   A user provides an explicit assessment of a recommendation. In our case, it would be a star rating, e.g., a user rates the movie four out of five stars.

Here, the recommendation problem will be viewed as a rating prediction problem.

2. **Implicit feedback:**

   Implicit feedback is extracted from a user's interaction with the recommended media. Most often, it is binary in nature. For instance, a user watched a movie (1), or they did not watch the movie (0).

Here, the recommendation problem will be viewed as a ranking problem.

One key advantage of utilizing implicit feedback is that it allows collecting a large amount of training data. This allows us to better personalize recommendations by getting to know our users more.

However, this is not the case with explicit feedback. People seldom rate the movies after watching them, as depicted by the funnel below.

The difference in data available to a system built on explicit feedback versus a system built on implicit feedback.

Explicit feedback faces the *missing not at random* (MNAR) problem. Users will generally rate those media recommendations that they liked. This means $\frac{4}{5}$ or $\frac{5}{5}$ star ratings are more common than $\frac{1}{5}$, $\frac{2}{5}$ or $\frac{3}{5}$. Therefore, we won't get much information on the kind of movies that a user does not like. Also, movies with fewer ratings would have less impact on the recommendation process.

Metrics

Let's look at the online and offline metrics used to judge the performance of the recommendation system.

We'll cover the following

- Types of metrics
- Online metrics
  - Engagement rate
  - Videos watched
  - Session watch time
- Offline metrics
  - mAP @ K
  - mAR @ K
  - F1 score
  - Offline metric for optimizing ranking

In this lesson, you will look at different metrics that you can use to gauge the performance of the recommendation system.

Types of metrics

Like any other optimization problem, there are two types of metrics to measure the success of a recommendation system.

1. **Online metrics**
   Online metrics are used to see the system's performance through online evaluations on live data during an A/B test.
2. **Offline metrics**
   Offline metrics are used in offline evaluations, which simulate the model's performance in the production environment.

We might train multiple models and tune and test these offline with the held-out test data (historical data). Only a subset of them will be promoted online. It is performance gain is worth the engineering effort to bring it into a production environment, the best performing model will then be selected for an online A/B test as for data.

Offline evaluation: Test model performance
Offline evaluation: Test model in production setup
Online evaluation: A/B Test
Live interactions between users and the model that is potentially model based and in online metrics
Model
Data
Model
Data
Model
Data

Architectural Components

Let's take a look at the architectural components of the recommendation system.

We'll cover the following

- Candidate generation
- Ranker
- Training data generation

It makes sense to consider generating the best recommendation from a large corpus of movies, as a multi-stage ranking problem. Let's see why.

We have a huge number of movies to choose from. Also, we require complex models to make great, personalized recommendations. However, if we try to run a complex model on the whole corpus, it would be inefficient in terms of execution time and computing resources usage.

Therefore, we split the recommendation task into two stages.

- Stage 1: Candidate generation
- Stage 2: Ranking of generated candidates

Stage 1 uses a simpler mechanism to sift through the entire corpus for possible recommendations. Stage 2 uses complex strategies only on the candidates given by stage 1 to come up with personalized recommendations.

Architectural diagram of the recommendation system

Candidate generation

Candidate generation is the first step in coming up with recommendations for the user. This component uses several techniques to find out the best candidate movies/shows for a user, given the user's historical interactions with the media and context.

> 📝 This component focuses on **higher recall**, meaning it focuses on gathering movies that might interest the user from all perspectives, e.g., media that is relevant based on historical user interests, trending locally, etc.

Ranker

The ranker component will score the candidate movies/shows generated by the candidate data generation component according to how interesting they might be for the user.

> 📝 This component focuses on **higher precision**, i.e., it will focus on the ranking of the top k recommendations.

It will ensemble different scores given to a media by multiple candidate generation sources whose scores are not directly comparable. Moreover, it will also use a lot of other dense and sparse features to ensure highly relevant and personalized results.

Training data generation

The user's engagement with the recommendations on their Netflix homepage will help to generate training data for both, the ranker component and the candidate generation component.

> 📝 Find out more about training data generation for the ranker component and the candidate generation component in the upcoming lessons.