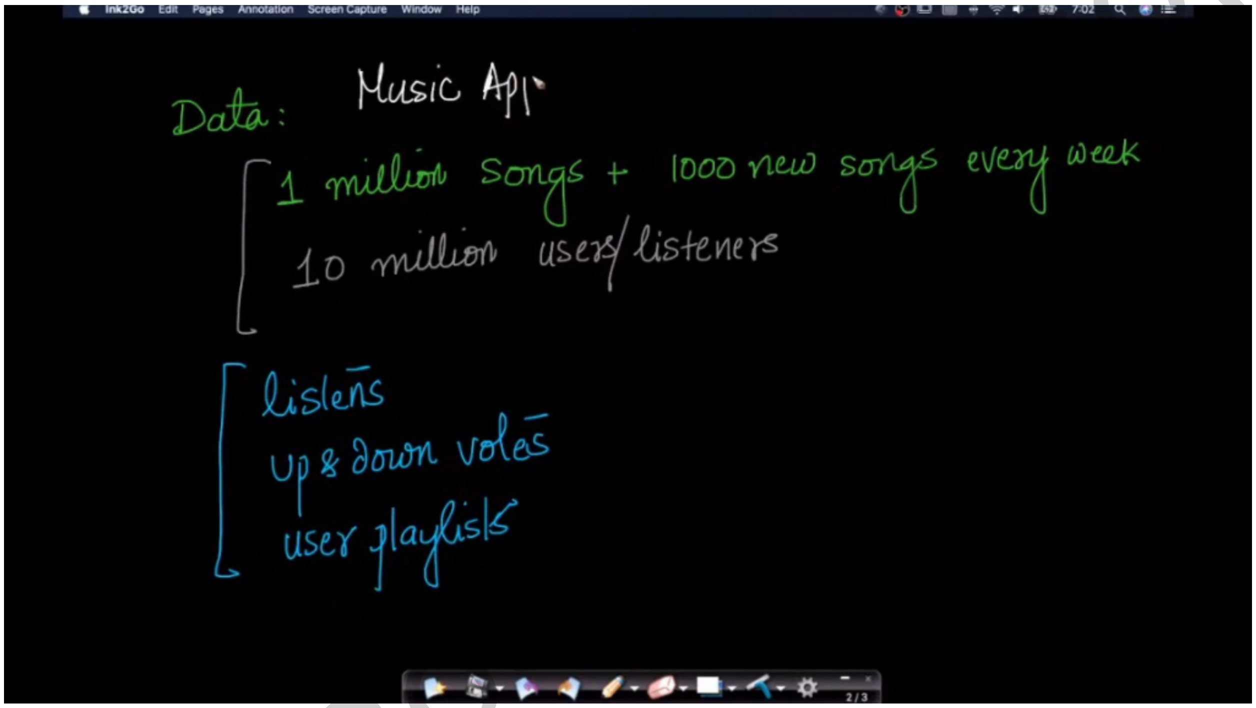
High Level + End-End Design of a Music Recommendation system - I & II

Data :



Data: Music App

1 million songs + 1000 new songs every week

10 million users/listeners

listens

up & down votes

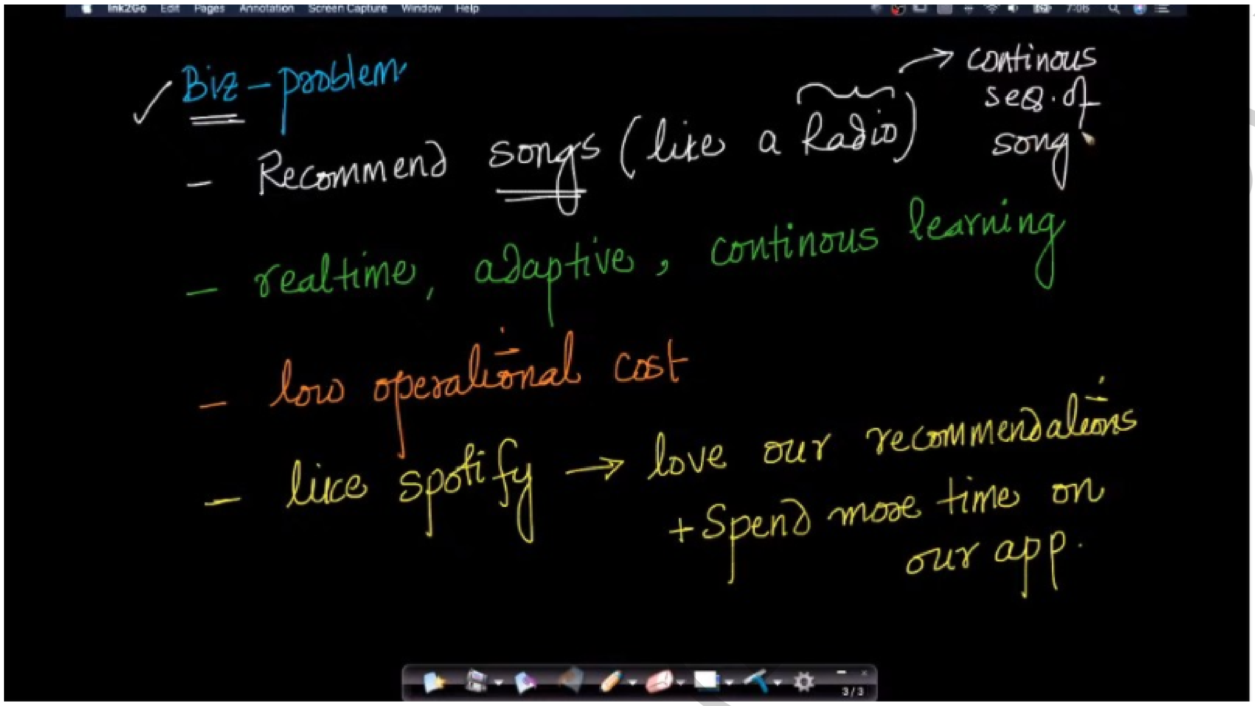
user playlists

Timestamp : 05:38

Assume we are dealing with a music streaming application.

1.) There would be around 1 million songs in the application database and also adding new songs every week.
2.) It has around 10 million users.
3.) Listens : User ui listen to song sj at time tij.
4.) Up and Down Votes : Each user can rate a song as +1 or -1 (Like or dislike).
5.) User playlists : Users can create their own playlist.

Business problem



✓ Biz-problem

- Recommend songs (like a Radio) → continous seq. of song.
- realtime, adaptive, continous learning
- low operational cost
- like spotify → love our recommendations + Spend more time on our app.

Timestamp : 09:01

1.) Recommend songs (like a radio).
2.) It should be in real time.
3.) It should be adaptive. The recommendation should consider the user preferences and also should consider the live data.
4.) Continuous learning.
5.) Low operational cost.
6.) It should be like spotify.

Business Metric :

The metric should be easily understandable and also must be reliable.

One such metric is : Number of recommended songs listened per user per day. This is a good metric to evaluate our system. If the value is high, then the recommendation is also good. It directly measures the performance of the system.

We can modify the above slightly and give a precise metric.

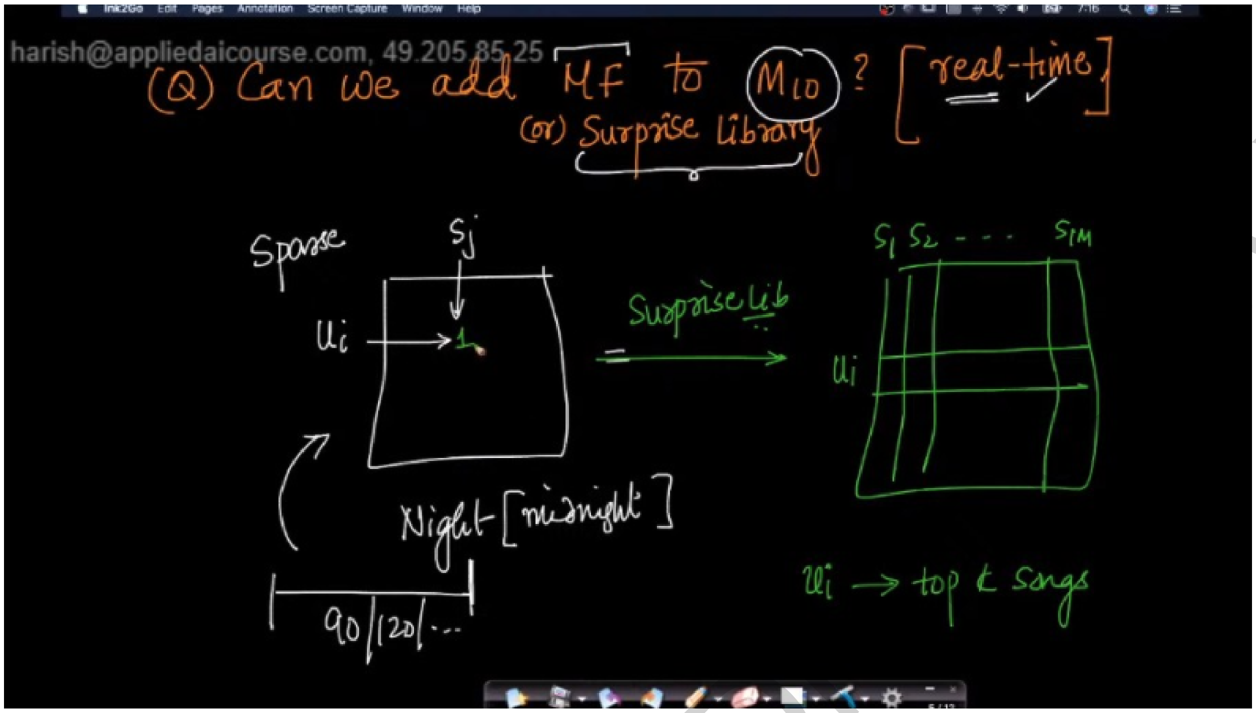
Percentage of recommended songs listened per user.

Pose

There are various approaches suggested by the students listening to the live session. One of the best solution a student suggested is as follows:

1.) Given a song in the audio format, extract the spectrogram from it.
2.) Using an autoencoder network, find a low dimensional latent representation of the spectrogram features i.e given a vector describing the spectrogram, use an autoencoder to transform this into a low dimensional representation and then learn back the original one.
3.) So, for each song, we can compute a vector.
4.) For the song-song similarity, we can use Elastic search for fast Nearest Neighbour search.
5.) For example, if a user listens to s1 and s2 (songs), then find all similar songs to s1 and s2. Take a union of them. Simply return the set of songs as the recommendation.
6.) If a user didn't listen to s3 but listened to s1 and s2, then we can perform this : sim(s1) U sim(s2) - sim(s3) where U represents Union and - represents the set difference.
7.) In this, we are not using much of the user's preferences. We are only considering the previous k songs listened by the user and computing the rest.

Can we incorporate matrix factorization to this ?



Timestamp : 19:42

1.) Every day at midnight, collect the data.
2.) Represent the data in the matrix given above. This will be a sparse matrix. This is because every user can't listen to every song. If a user has listened to a song, then fill it with 1. If a user skipped it, fill it with -1. If a user hasn't listened to the song yet, fill it with 0.
3.) Using functions from the surprise library, fill the matrix.
4.) Rather than storing the entire matrix, for each user ui stores only the top k entries.

What strategy would you use to recommend popular songs ?

You can use max heap data structure to retrieve the top k songs which by definition is popular based on the average listen time.

What about the cold start problem?

It can happen to both the user and song. For example, a new user may join for which we don't have data. For this, we can use the location of the user and then check all those users in that location and then recommend the song. This is one of the solutions. You can think of many. You can consider age,gender and many other factors.

For songs, you can simply pass the spectrogram of the new song to the autoencoder network. It will give you a latent vector. Using this, we can find similar songs.

End-End System

1.) **Data processing pipeline :**

1.1) Model 1 (Song-song similarity) : As soon as a song arrives, pass it to the autoencoder network and get the vector representation. You can use this to compute the similarity and other things.

1.2) Model 2 (Matrix Factorization) : This is a nightly job. You can use the surprise library.

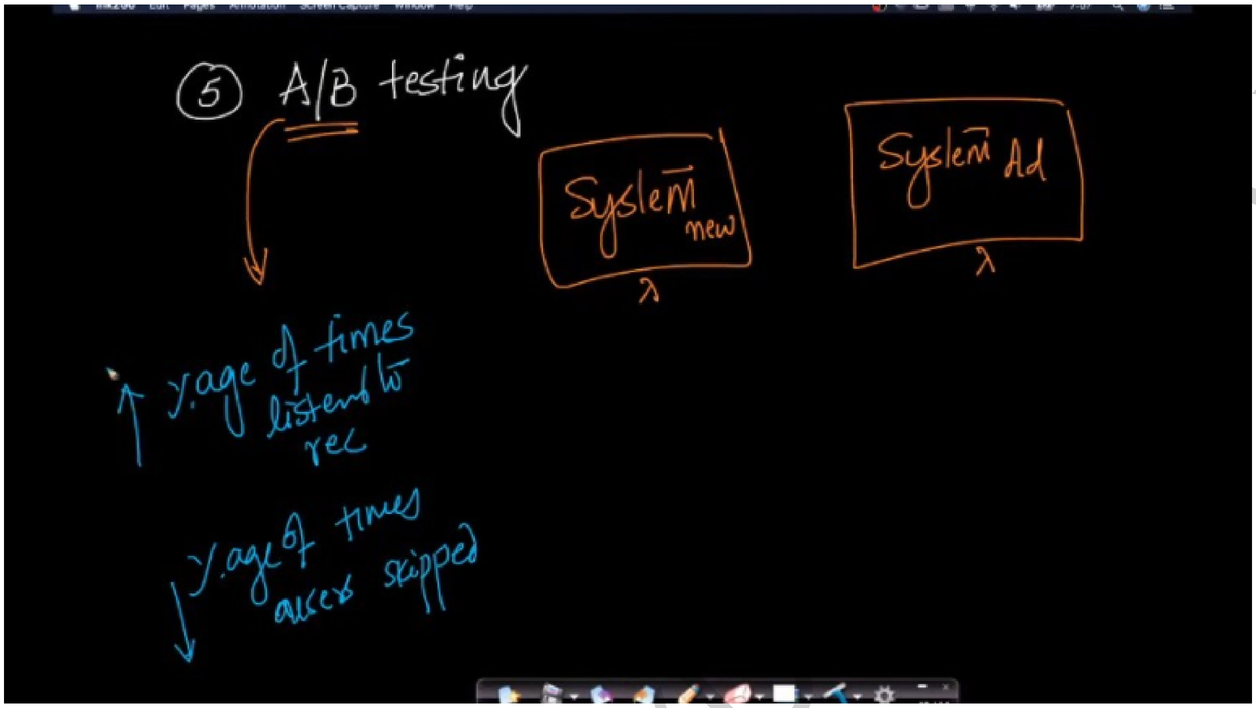
1.3) Popularity based System : You can use a hash table.

2.) **Modelling :** For deep learning based methods, you can use autoencoders. For the matrix factorization, you can use the functionalities from the surprise library. For popularity based systems, you can implement your own hash table.

3.) **Runtime System :** We can use flask or rest api's.

4.) **Model updation :** For model 1, we can update in real-time. For model 2, we need to make sure it's updated nightly. For model 3, we need to continually update it.

5.) **A/B testing :**



⑤ A/B testing
System new
System old
↑ %age of times listened to rec
↓ %age of times users skipped

Timestamp : 01:00:51

We can randomly split by users and then compute the A/B testing.

Keep 80% of the traffic in the old system itself. Send the 10% of the traffic to the old system itself. The remaining 10% is sent to the new system. Run A/B test.

Let's consider the metric % of times users listened to the recommended song. Initially, this would fluctuate. Then after some time, it will roughly stabilize to a value. Keep track of this metric value in the above split. Then compare the stabilized value.

6.) **Model maintenance** : Over a period of time, if the desired performance is not achieved, then we need to analyze our model. For this, we need to keep track of the metric over time.

There are multiple approaches for this problem. Please try using other approaches too.

Note : In the live session, multiple approaches to this problem have been discussed. In these notes, we have given the description of best approaches.