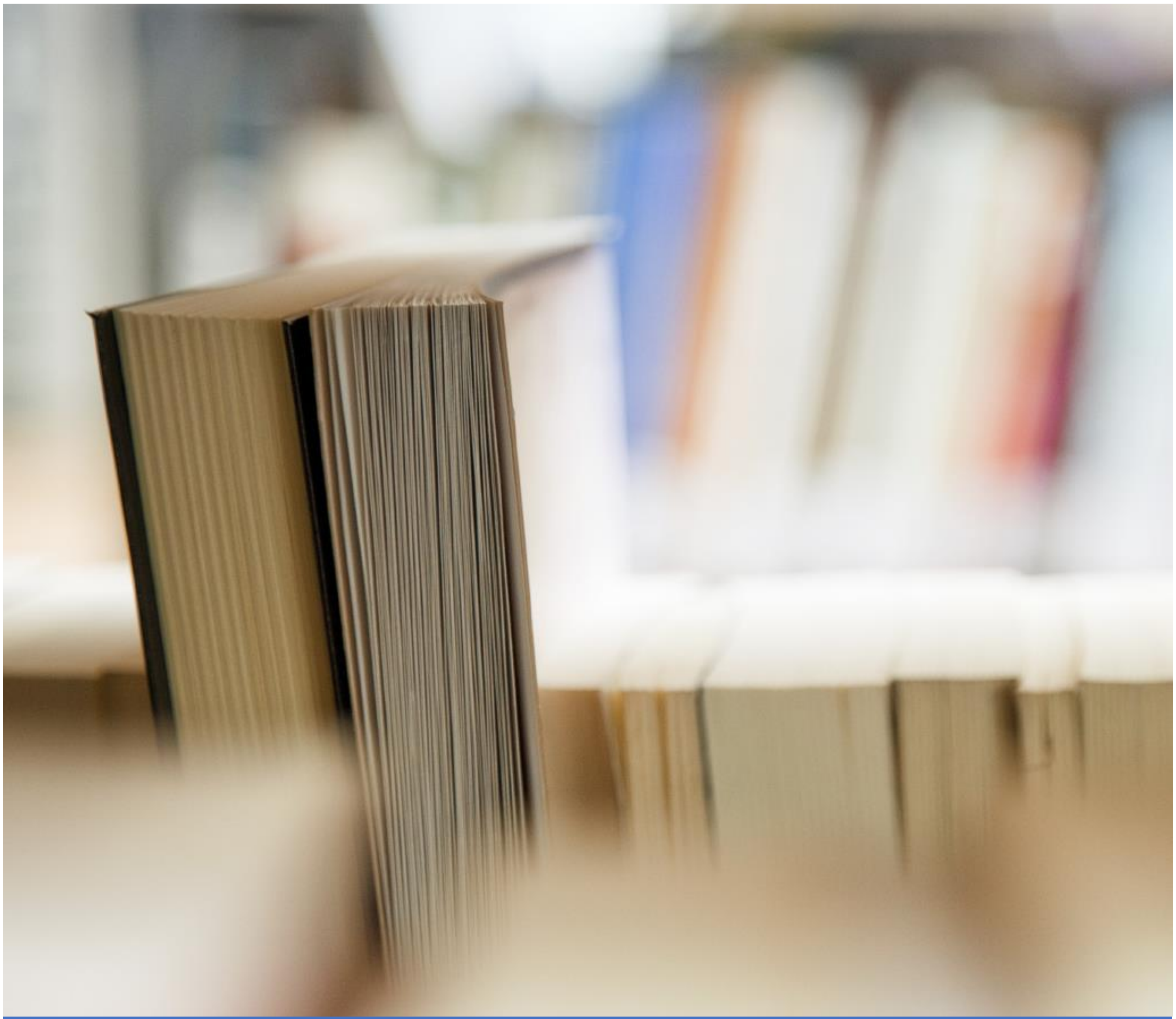


FRIDAY

Final Paper

Deepali Pareek, Rahul Sharma, Shruthi Jabshetty, Anubhav Pradhan

5/4/19

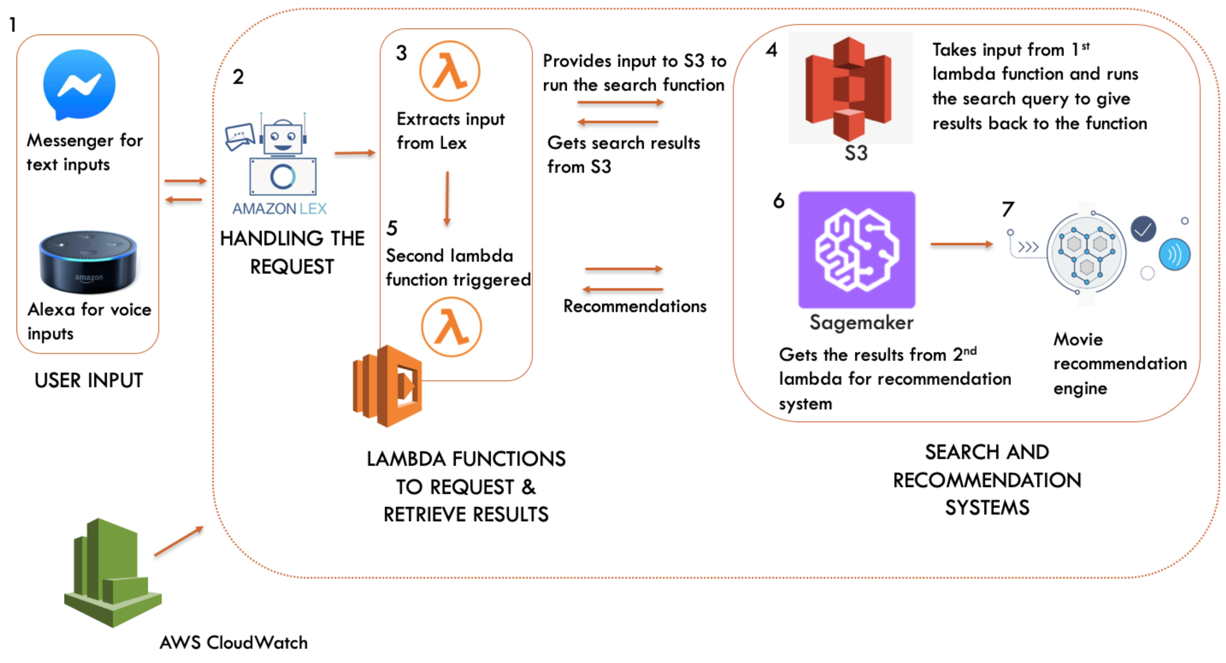
Big data Analytics Experience

Contents

Introduction:

FRIDAY at core is a learning chatbot that can take any dialogue as an input, either in the form of text or voice and can return the name of the movie to which the dialogues belongs to, also based on the name of the movie it will also find suitable recommendation of similar movies

The product architecture:


1
Messenger for text inputs
Alexa for voice inputs
USER INPUT
2
AMAZON LEX
HANDLING THE REQUEST
3
Extracts input from Lex
5
Second lambda function triggered
LAMBDA FUNCTIONS TO REQUEST & RETRIEVE RESULTS
Provides input to S3 to run the search function
Gets search results from S3
Recommendations
4
S3
Takes input from 1st lambda function and runs the search query to give results back to the function
6
Sagemaker
Gets the results from 2nd lambda for recommendation system
7
Movie recommendation engine
SEARCH AND RECOMMENDATION SYSTEMS
AWS CloudWatch

- Quick description:
  - The system currently takes text inputs, integrated with Facebook messenger.
  - AWS Lex handles the input
  - AWS LAMBDA for computing
  - Use AWS Dynamo DB for persistent data storage
  - Use AWS Sagemaker to implement Movie Recommendation
  - AWS CloudWatch to take into account the usage of AWS services.

Data extraction and loading:

ETL provides the underlying infrastructure for integration by performing three important functions:

- Extract: Read data from the source database.

- Transform: Convert the format of the extracted data so that it conforms to the requirements of the target database. Transformation is done by using rules or merging data with other data.
- Load: Write data to the target database.

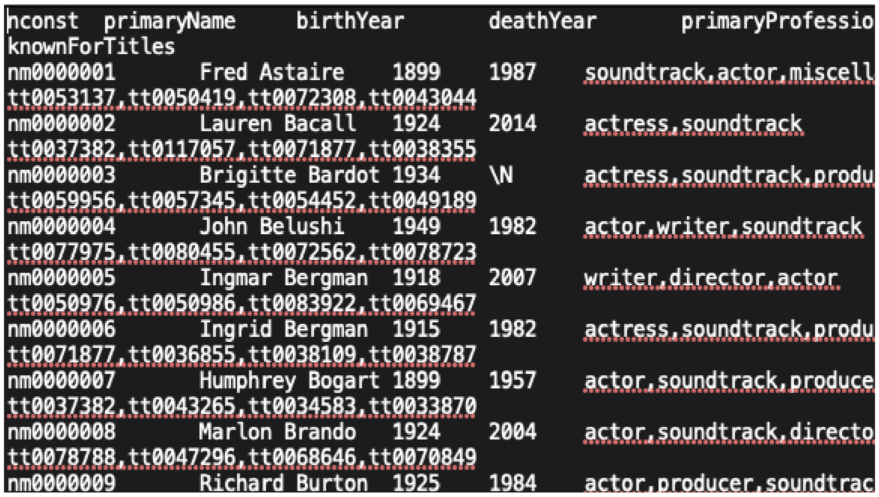
Given the growth and importance of unstructured data for decision making, ETL solutions from major vendors are beginning to offer standardized approaches to transforming unstructured data so that it can be more easily integrated with operational structured data.

Data Details:

The data that we have in hand are primarily flat files in different formats. The scripts are in text and the Movie details, actors' details and ratings are tab separated files and text files. For querying the dialogues, we need the files in same formats which should also support our chatbot model.
The data is roughly over 2 GB and we would need to load and transform it in a way that would not slow down the model in any way.

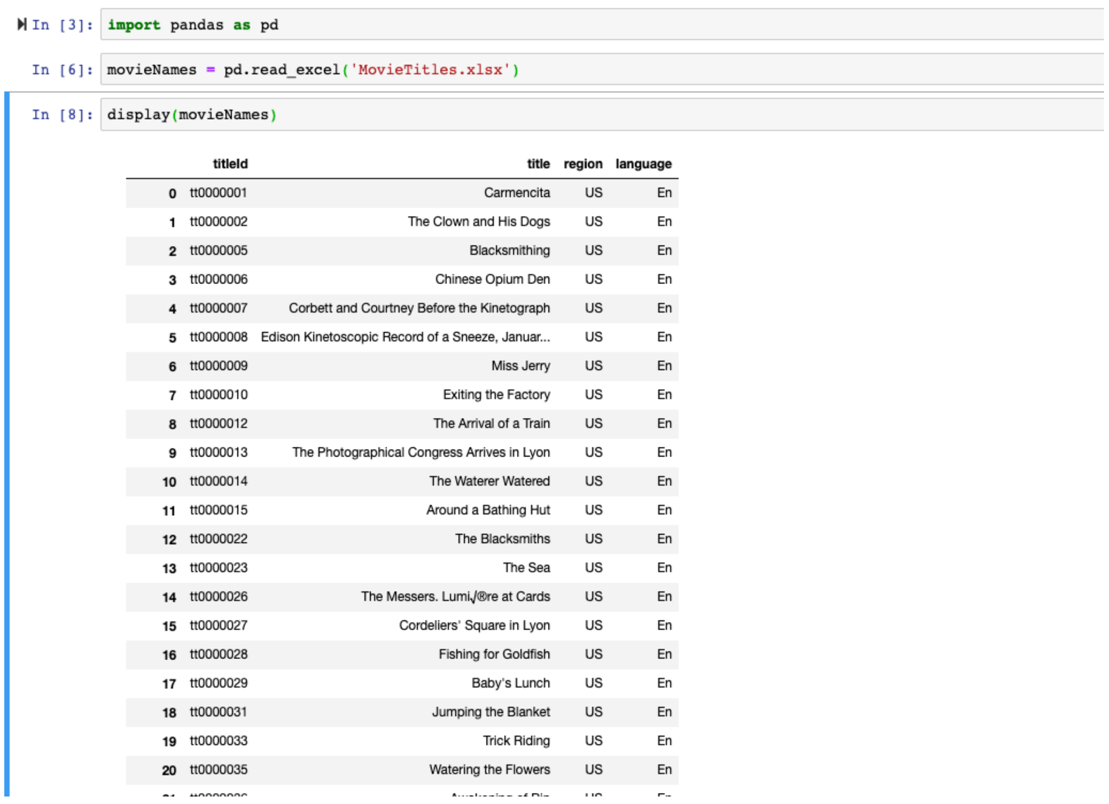
Approach taken:

The current files would be taken in dataframes to enable us to do further transformations and querying.
Below are some snippets of current data format:



```
nconst  primaryName     birthYear       deathYear       primaryProfessio
knownForTitles
nm0000001       Fred Astaire    1899    1987    soundtrack,actor,miscell
tt0053137,tt0050419,tt0072308,tt0043044
nm0000002       Lauren Bacall   1924    2014    actress,soundtrack
tt0037382,tt0117057,tt0071877,tt0038355
nm0000003       Brigitte Bardot 1934    \N      actress,soundtrack,produ
tt0059956,tt0057345,tt0054452,tt0049189
nm0000004       John Belushi    1949    1982    actor,writer,soundtrack
tt0077975,tt0080455,tt0072562,tt0078723
nm0000005       Ingmar Bergman  1918    2007    writer,director,actor
tt0050976,tt0050986,tt0083922,tt0069467
nm0000006       Ingrid Bergman  1915    1982    actress,soundtrack,produ
tt0071877,tt0036855,tt0038109,tt0038787
nm0000007       Humphrey Bogart 1899    1957    actor,soundtrack,produce
tt0037382,tt0043265,tt0034583,tt0033870
nm0000008       Marlon Brando   1924    2004    actor,soundtrack,directo
tt0078788,tt0047296,tt0068646,tt0070849
nm0000009       Richard Burton  1925    1984    actor,producer,soundtrac
```

To change the format of this data, we use python scripts and load these flat files into dataframes. We take all the files and convert them to dataframes before storage.
Here is a snippet of how the dataframe looks:



```
In [3]: import pandas as pd

In [6]: movieNames = pd.read_excel('MovieTitles.xlsx')

In [8]: display(movieNames)
```

	titleId	title	region	language
0	tt0000001	Carmencita	US	En
1	tt0000002	The Clown and His Dogs	US	En
2	tt0000005	Blacksmithing	US	En
3	tt0000006	Chinese Opium Den	US	En
4	tt0000007	Corbett and Courtney Before the Kinetograph	US	En
5	tt0000008	Edison Kinetoscopic Record of a Sneeze, Januar...	US	En
6	tt0000009	Miss Jerry	US	En
7	tt0000010	Exiting the Factory	US	En
8	tt0000012	The Arrival of a Train	US	En
9	tt0000013	The Photographical Congress Arrives in Lyon	US	En
10	tt0000014	The Waterer Watered	US	En
11	tt0000015	Around a Bathing Hut	US	En
12	tt0000022	The Blacksmiths	US	En
13	tt0000023	The Sea	US	En
14	tt0000026	The Messers. Lumi√®re at Cards	US	En
15	tt0000027	Cordeliers' Square in Lyon	US	En
16	tt0000028	Fishing for Goldfish	US	En
17	tt0000029	Baby's Lunch	US	En
18	tt0000031	Jumping the Blanket	US	En
19	tt0000033	Trick Riding	US	En
20	tt0000035	Watering the Flowers	US	En

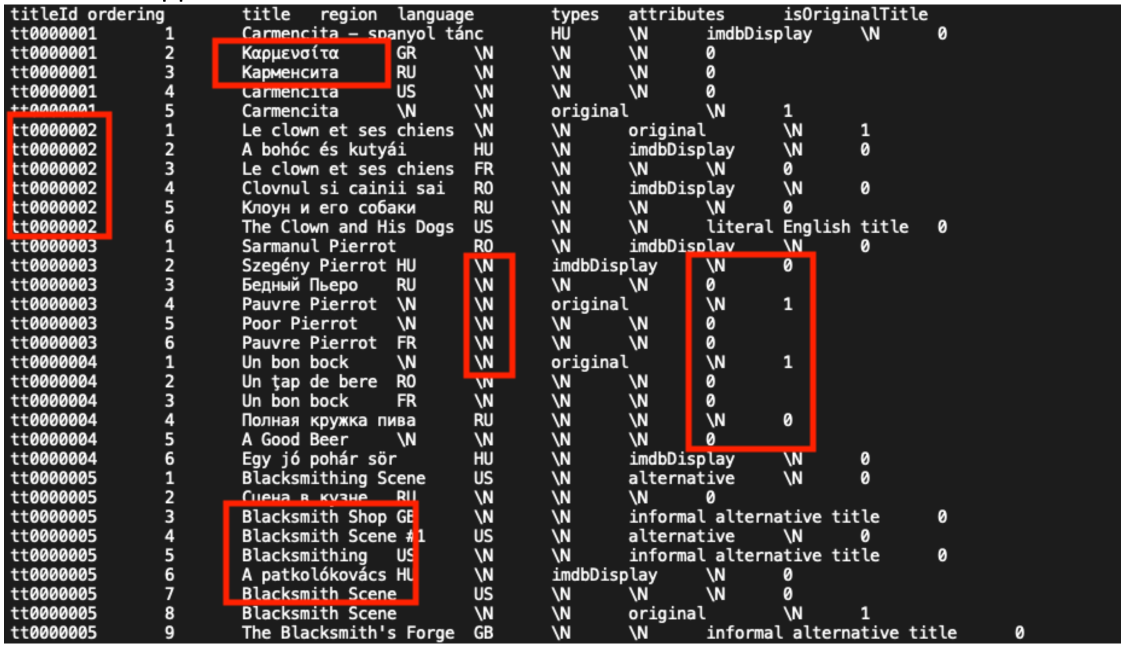
Data cleaning and transformation:

Problems faced:

- **Missing Data**: The .tsv files which contained the details such as movie titles, actor details, IMDB ratings etc were all having missing data. There were \NA\ values in various fields which would eventually create a problem while querying this data.
- **Data Inconsistency**: The data format was incoherent and inconsistent. There were no primary keys, indexing would become difficult at later stage because of missing primary Ids.
- **Duplicate values:** The files also had duplicated values for various fields like Movie titles. There were multiple rows for same movie. This would lead to an inefficient search system.
- **Data errors:** The fields with movie titles of different regions had the title fields encrypted and the language could not be deciphered. These values made no sense.
- **Unnecessary Columns:** There were columns that were of no use during movie search or recommendation. These columns would only increase the load on the system while querying hence making the product inefficient.

- **Non matching data format:** The datasets with different movie details had different formats. For example, the movie name in “Titles” dataset was all lowercase and without space, while the movie names in dataset for recommendation engine had Camel case, with space and movie year included.

Here is a snippet of a file with movie titles:



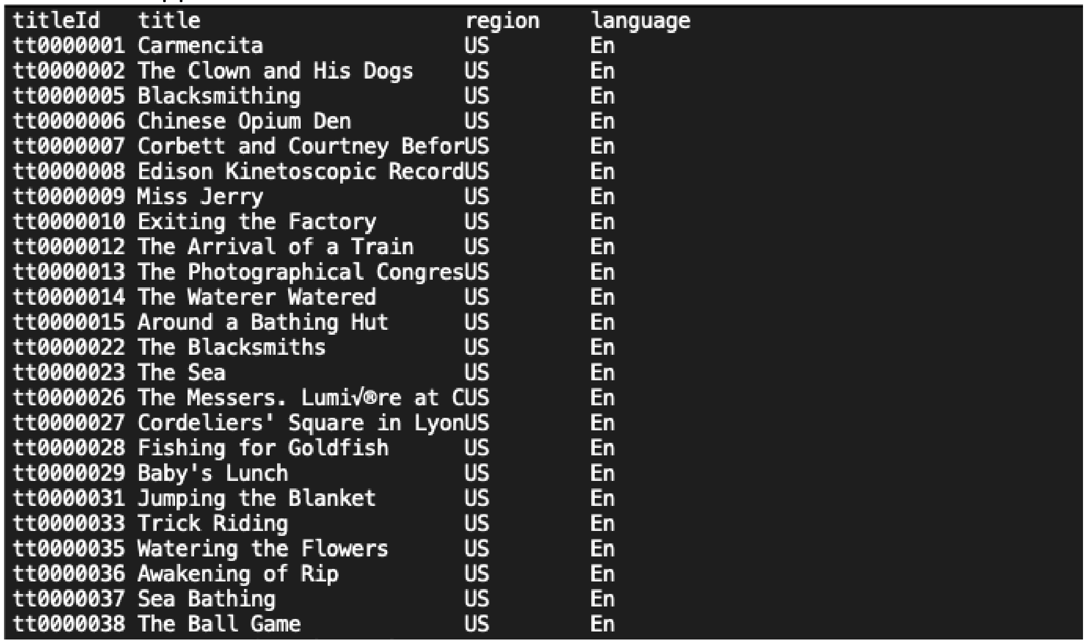
```
titleId ordering        title   region  language        types   attributes      isOriginalTitle
tt0000001       1       Carmencita - spanyol tánc       HU      \N      imdbDisplay     \N      0
tt0000001       2       Καρμενσίτα      GR      \N      \N      \N      0
tt0000001       3       Карменсита      RU      \N      \N      \N      0
tt0000001       4       Carmencita      US      \N      \N      \N      0
tt0000001       5       Carmencita      \N      \N      original        \N      1
tt0000002       1       Le clown et ses chiens  \N      \N      original        \N      1
tt0000002       2       A bohóc és kutyái       HU      \N      imdbDisplay     \N      0
tt0000002       3       Le clown et ses chiens  FR      \N      \N      \N      0
tt0000002       4       Clovnul si cainii sai   RO      \N      imdbDisplay     \N      0
tt0000002       5       Клоун и его собаки      RU      \N      \N      \N      0
tt0000002       6       The Clown and His Dogs  US      \N      \N      literal English title   0
tt0000003       1       Sarmanul Pierrot        RO      \N      imdbDisplay     \N      0
tt0000003       2       Szegény Pierrot HU      \N      imdbDisplay     \N      0
tt0000003       3       Бедный Пьеро    RU      \N      \N      \N      0
tt0000003       4       Pauvre Pierrot  \N      \N      original        \N      1
tt0000003       5       Poor Pierrot    \N      \N      \N      \N      0
tt0000003       6       Pauvre Pierrot  FR      \N      \N      \N      0
tt0000004       1       Un bon bock     \N      \N      original        \N      1
tt0000004       2       Un ţap de bere  RO      \N      \N      \N      0
tt0000004       3       Un bon bock     FR      \N      \N      \N      0
tt0000004       4       Полная кружка пива      RU      \N      \N      \N      0
tt0000004       5       A Good Beer     \N      \N      \N      \N      0
tt0000004       6       Egy jó pohár sör        HU      \N      imdbDisplay     \N      0
tt0000005       1       Blacksmithing Scene     US      \N      alternative     \N      0
tt0000005       2       Сцена в кузне   RU      \N      \N      \N      0
tt0000005       3       Blacksmith Shop GB      \N      \N      informal alternative title      0
tt0000005       4       Blacksmith Scene #1     US      \N      alternative     \N      0
tt0000005       5       Blacksmithing   US      \N      \N      informal alternative title      0
tt0000005       6       A patkolókovács HU      \N      imdbDisplay     \N      0
tt0000005       7       Blacksmith Scene        US      \N      \N      \N      0
tt0000005       8       Blacksmith Scene        \N      \N      original        \N      1
tt0000005       9       The Blacksmith's Forge  GB      \N      \N      informal alternative title      0
```

Solutions implemented:

DATA PROBLEMS	SOLUTIONS
MISSING DATA	The columns which had maximum null values were excluded starting the process, then the rows with null values has to checked and had to be adjusted using logic or had to be excluded from the data. Example- If the language had null value for a movie of US as a region, the language was set to English. And if the Language, region and movie title were all null, the row had to be excluded.
INCONSISTENT DATA	For the inconsistency in data, the major problem was the missing primary key and the repetition of the Movie title ID. We had to deal with the title ID by using ‘GroupBy’ clause, doing this grouped all the titles with same ID together and there were no repeating values in the ID column. This also solved the problem of missing primary key.

DUPLICATE DATA	Similar movie names, titles denoting one movie were all merged to avoid confusion in the movie title and the movie ID.
DATA ERRORS	The encrypted values would be of no use during querying dialogues, as we would just be querying English dialogues, hence the encrypted and different language titles had to be excluded.
EXTRANEOUS DATA	The data which has no impact on our model would only make it slow. Hence extra data was eliminated.
NON MATCHING VALUES	A script was created to load the data in similar formats. All lower without spaces.

Here is a snippet of how the same file looks now:



```
titleId   title                      region    language
tt0000001 Carmencita                 US        En
tt0000002 The Clown and His Dogs     US        En
tt0000005 Blacksmithing              US        En
tt0000006 Chinese Opium Den          US        En
tt0000007 Corbett and Courtney BeforUS        En
tt0000008 Edison Kinetoscopic RecordUS        En
tt0000009 Miss Jerry                 US        En
tt0000010 Exiting the Factory        US        En
tt0000012 The Arrival of a Train     US        En
tt0000013 The Photographical CongresUS        En
tt0000014 The Waterer Watered        US        En
tt0000015 Around a Bathing Hut       US        En
tt0000022 The Blacksmiths            US        En
tt0000023 The Sea                    US        En
tt0000026 The Messers. Lumi√®re at CUS        En
tt0000027 Cordeliers' Square in LyonUS        En
tt0000028 Fishing for Goldfish       US        En
tt0000029 Baby's Lunch               US        En
tt0000031 Jumping the Blanket        US        En
tt0000033 Trick Riding               US        En
tt0000035 Watering the Flowers       US        En
tt0000036 Awakening of Rip           US        En
tt0000037 Sea Bathing                US        En
tt0000038 The Ball Game              US        En
```

Key takeaways:

There are several types of data quality problems and before doing any kind of analysis it's important that we take into account all the quality concerns. With our data, we faced various data quality issues and we tried to solve them step by step, following were the key takeaways for us-

1. Missing attributes and blank fields: If blank fields are identified is there a meaning behind those values which we'd want to explore before eliminating all those rows or columns.
2. Spot checks: While the data might appear clean it is important to make sure that we spot check the data. While doing so in this data set, we could find spelling errors that might have caused problems in later querying and transformations.

3. Noise Vs phenomena: The incoherent fields might not be noise all the time, in our case, we found that tile field had some encrypted data and that led us to realizing that we might want to focus on the movies belonging to one particular region.
4. Plausibility check: These checks enabled us to find any apparent conflicts such as a movie from a region has a language of different region.
5. Relevancy of Data: A field might consistent, error free and might look harmless, but if it is not making any impact on the model, we'd want to exclude the field.

Data Loading/Storage

For data storage we have gone through multiple options and after analyzing the pros and cons of multiple storage, data retrieval and compute options we have decided to move ahead with AWS S3 (Simple storage Service) and LAMBDA. Mentioned below are the details of the challenges we have faced in each approach and how have we decided on our primary storage.

Approach 1 - AWS-DynamoDB

Dynamo DB is a key-value and document database, which delivers millisecond performance at any scale. It is a fully managed AWS service which has in memory caching capabilities and suitable for internet scale applications

Reason why we considered Dynamo DB

- The movie data set that we have, which contains the name of the movie and the complete script for the movie is primarily in TEXT format
- Dynamo DB is a no-SQL database which is suitable for the storage of unstructured data and when combined with Dynamo DB accelerator (In -Memory Caching) gives a higher throughput for every query made
- Dynamo-DB is a fully scalable and managed database solution provided by AWS, which would automatically address, any scalability and availability related concern which might raise in the future
- Dynamo-DB allows the use of Local Secondary Indexes (LSI) and Global Secondary Indexes (GSI), with the help of LSI and GSI, we would have the capabilities to set up indexes based on our requirement and that would efficient and customizable way of querying the data

Why storage in Dynamo-DB is not suitable for our Purpose

- The Movie data available with us contains two fields primarily, the movie name and the entire movie script

- The primary search parameter that we would receive from the front end is a string of movie dialogue. The string would contain no further information about any other field which could be used a key, basis which a document search could be performed
- The lack of any Key that could be passed along with the dialogues and the huge text size of the Movie script field, made Dynamo DB an inefficient storage option for us

Approach 2 - Simple Storage Services(S3) & Athena or Elastic Search

AWS S3 is a performant, secure, highly available and durable object storage service. S3 allows unlimited object data storage.
Athena is an interactive query service that makes it extremely easy to analyze data stored in S3 with the help of standard SQL
AWS elastic Search is a fully managed service that makes it s easier to deploy, secure and operate elastic search at scale and zero downtime

Reason why we considered Simple Storage Services(S3) & Athena

- S3 is a cheap, durable and available object storage service and is well suited for storing large amount of data
- Provisioning, Integration and maintenance of S3 buckets is very easy
- S3 comes with a variety of security features which can be easily incorporated and allows building of secured applications
- Athena is a serverless interactive query service, which is very easy to provision and use
- With Athena we simply have to point it to our data and S3, define our schema and it would allow us to perform query on the underlying data
- This is a completely scalable solution and would require no manual intervention to address any scalability related issues in future. Primarily because S3 provides unlimited storage
- Elastic Search allows to perform Full Text Search on Character Large Object Type Data (CLOB- They help store large character data in database and enable simple query actions like LIKE on them)

Why storage in Simple Storage Services(S3) and using Athena or Elastic Search to query is not suitable for our Purpose

- The storage of entire script of a movie in a relational database field would require special datatype as it would violate the norms of normal string storage
- Storing the entire script into a relational database schema would require us to store the script data in Character Large Object (CLOB) format. CLOB helps store large amount of character data (Up to 4 GB). It is similar to Binary large object (BLOB) but uses character encoding in place of binary encoding. CLOB allows simple character operations LIKE.

- In order to search a particular Dialogue from the entire script column and then retrieve the relevant movie name would require us to perform a full text search (FTS) and Athena doesn't not support full text search
- Athena doesn't support CLOB data type
- Using elastic Search would require us to provision additional elastic search clusters, which would incur additional cost in the long run. We have found a more efficient and cost-effective method to store and retrieve data

Approach 3 - Simple Storage Services(S3) & LAMBDA function

AWS Lambda is a compute service, where once the code is uploaded, it takes care of the , scaling, patching and administration for the infrastructure required to run the applications and also monitors performance by publishing real time logs.

Reason why we considered Simple Storage Services(S3) & LAMBDA function is the most suitable storage and compute option for our case

- One of the primary data set (Movie Script Dataset) that we have at our hand has three things
    - Movie name as part of the file name
    - The Movie name as part of the content
    - Script for the movie as part of the content
- S3 serves as an excellent source for data storage for the raw data files (Details of which has been discussed in the previous section)
- Our search input would be a dialogue (Which will be a string) and expected output would be the name of the movie to which the dialogue belongs
- We plan to use Lambda function, which once invoked would perform the following steps
    - Take the Movie Dialogue as an input
    - Perform a grep operation to search the file name that contain the input string
    - Parse the entire file name and select the movie name from it
    - Return the movie name as an output

- Using Lambda function to perform a GREP operation is most efficient because of the following reasons
    - GREP performs in memory computation where it does not use the conventional line-oriented search, instead it reads the raw data into a large buffer file
    - It avoids looking at every input byte
    - Executes very few instructions for each byte that it does look
    - It uses the Boyer-Moore algorithm to search effectively and it avoids writing of data and also avoids breaking the input into multiple lines of string.

Data Visualization

Data visualization is the presentation of data in a pictorial or graphical format. It enables decision makers to see analytics presented visually, so they can grasp difficult concepts or identify new patterns.

Overview of the Data Set We are Using

Dataset	Number of Records	
Actors	10 Million	
Movie Reviews & Ratings	9 Million	
Movie Titles	10 Million	
Movie Scripts per Genre	Action	259
	Adventure	139
	Animation	28
	Biography	4
	Comedy	312
	Crime	188
	Drama	540
	Family	29
	Fantasy	86
	Film-Noir	5
	History	4
	Horror	127
	Music	6
	Musical	19
	Mystery	99
	Romance	175
	Sci-Fi	129
	Short	2
	Sport	2
	Thriller	330
	War	23
	Western	14

There are two major avenues where we think, incorporating visualization would further enhance our analytical capabilities. Mentioned below are the details of the two avenues and relevant visualizations

- Visualizations related to the Recommendation Engine (Movie Ratings is one of the primary criteria basis which the recommendation are being made) ○ Number of Ratings Histogram ○ Distribution of Ratings ○ Normalization of Data
    - Relationship betweeen Actual average rating and number of ratings
- Visualizations related to the Chatbot performance ○ Active Conversation Period
    - Type of Conversations (new conversation vs total conversation) ○ Retention Rate

Visualizations related to the Recommendation Engine

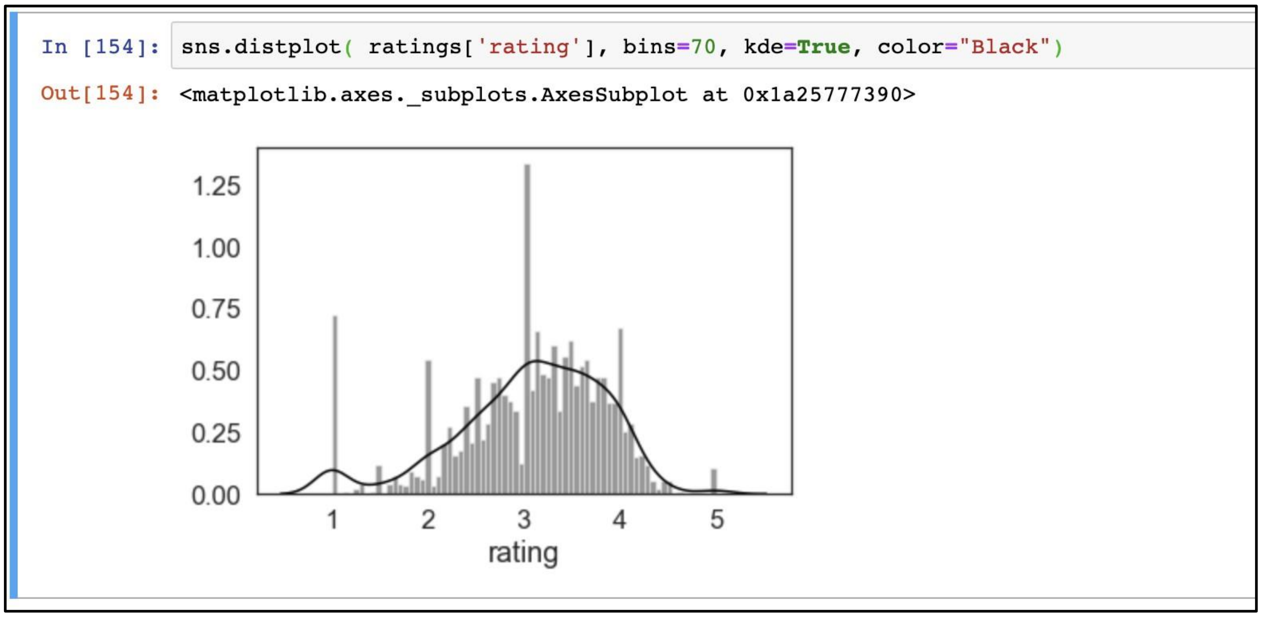
We are primarily using two different types of recommendation engine

- user-user based, and item-item based collaborative filtering to make movie recommendations from users' ratings data.
- low-rank matrix factorization. We are using Singular Vector Decomposition (SVD)

Distribution of Ratings

We can notice some peaks on whole numbers. This makes sense as people tend to give whole number ratings. The valleys can also be visibly noticed as 1.25, 3.45 stars are not usually given ratings. Most of the movies are distributed a little but normally with the center of distribution at around 3 stars/ 3.5 stars.

This histogram also shows the kernel density plot. The kde is little bit normal and affected by the un usual ratings.
There are some outliers of one-star movies for the bad movies and a little peak at five stars, this could be a popular movie, or a movie watched by a single person.

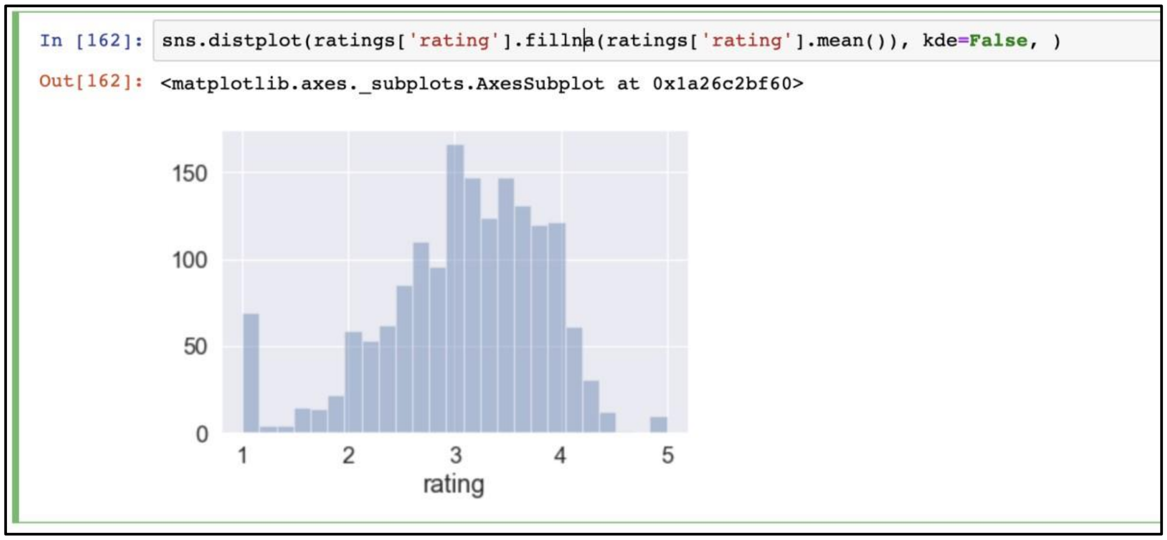


```
In [154]: sns.distplot( ratings['rating'], bins=70, kde=True, color="Black")
Out[154]: <matplotlib.axes._subplots.AxesSubplot at 0x1a25777390>
```

Normalizations of Rating Distributions

As the above distribution is somewhat normal, we can replace the NA values in this with the mean of the distribution and see how it varies over 1 to 5 scale.

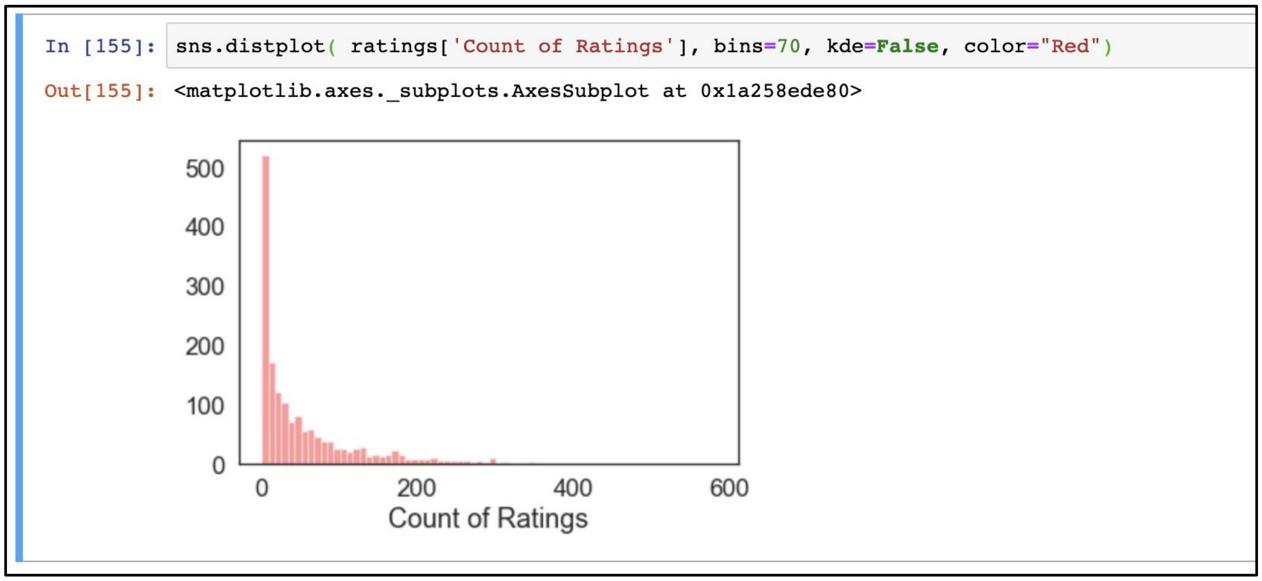
After replacing in with the mean value, the distribution becomes more normal than with the NA values.



```
In [162]: sns.distplot(ratings['rating'].fillna(ratings['rating'].mean()), kde=False, )
Out[162]: <matplotlib.axes._subplots.AxesSubplot at 0x1a26c2bf60>
```

Number of Ratings Histogram

This histogram plots the frequency of number of ratings. As we can see that, most of the number of ratings are quite few i.e. Most movies have Zero or One rating. This makes sense as most people watch only famous movies or blockbusters.

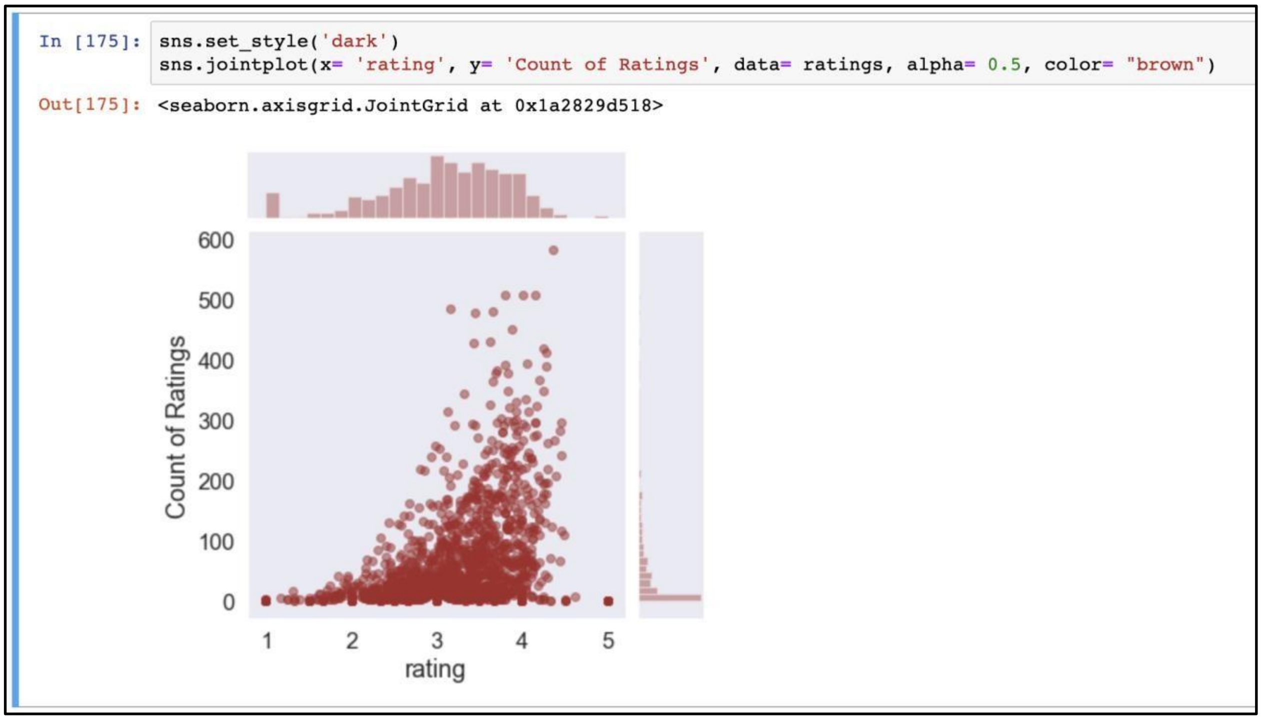


```
In [155]: sns.distplot( ratings['Count of Ratings'], bins=70, kde=False, color="Red")
Out[155]: <matplotlib.axes._subplots.AxesSubplot at 0x1a258ede80>
```

Relationship betweeen Actual average rating and number of ratings

We can clearly see that as a movie gets a greater number of ratings, that movie is more likely to have higher average rating. This makes sense as better the movie, more people tend to watch it, more are the reviews given. One- or two-star movies have very few counts and five-star ratings are outliers as not many people give five-star ratings to the movies.

This scatter plot is positively and, in some way, highly correlated. The correlation for this can be observed using the Pearson correlation coefficient.

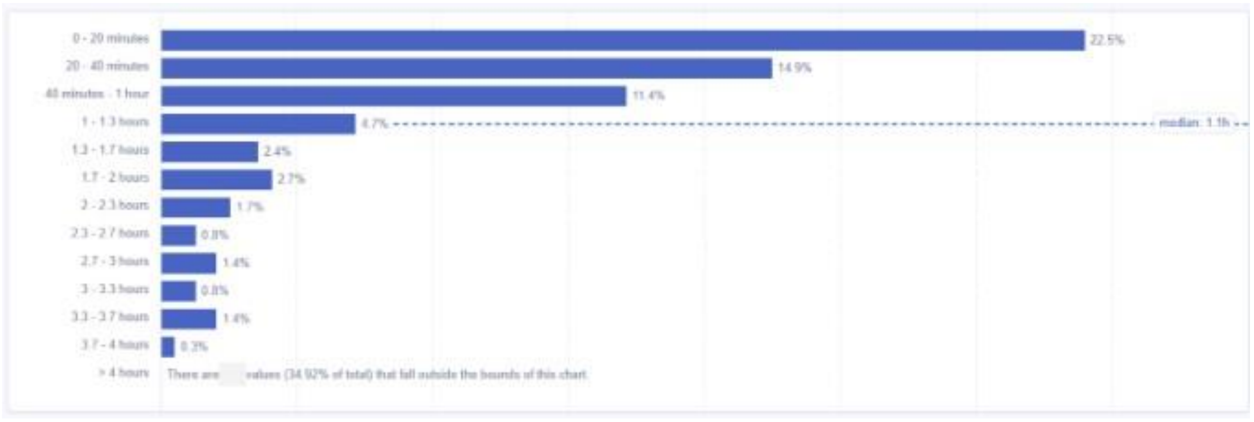


```
In [175]: sns.set_style('dark')
          sns.jointplot(x= 'rating', y= 'Count of Ratings', data= ratings, alpha= 0.5, color= "brown")

Out[175]: <seaborn.axisgrid.JointGrid at 0x1a2829d518>
```

Visualizations related to the Chatbot performance

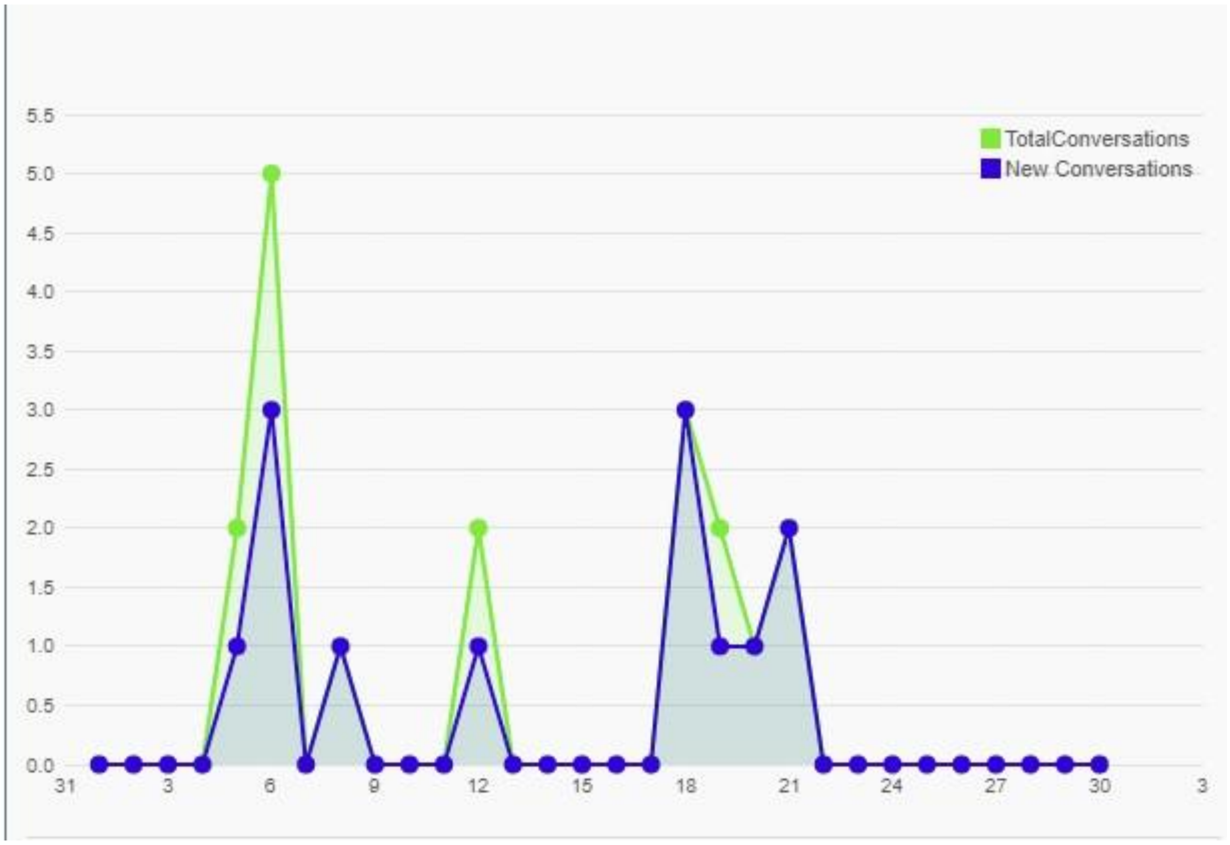
Active Conversation Period

There is a general perception that the more the number of user using your chatbot the better your chatbot is, but the number of active users who regularly and successfully use your chatbot is a more tangible measure as it gives a better insight towards your real user experience. In order to track the active conversation period, we plan to keep of the total time period an user is engaged in conversation with our chatbot



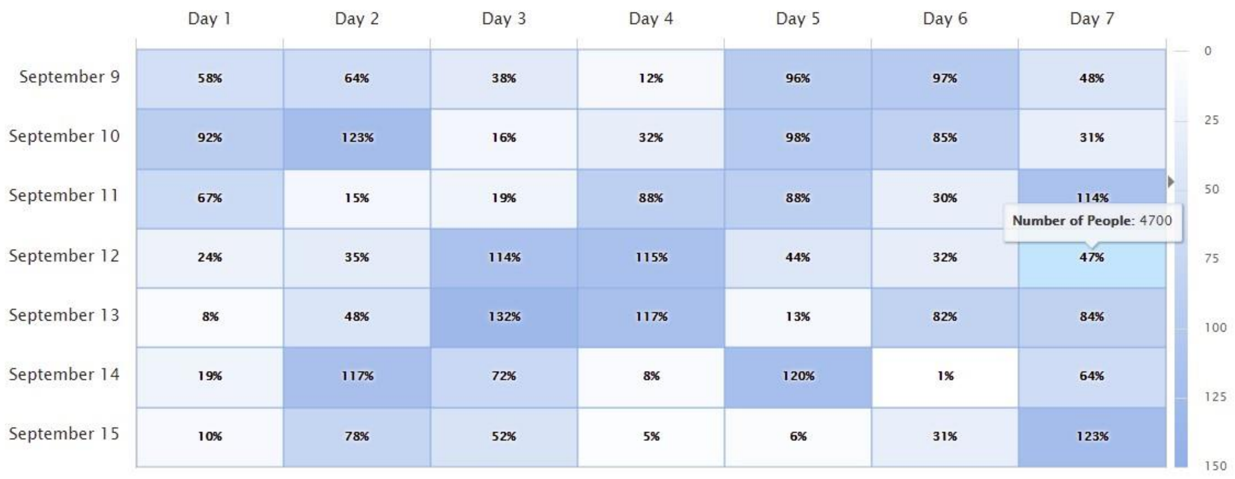
Type of Conversations (new conversation vs total conversation)

The number of different conversation (new conversation and total conversation) that we have with the chatbot is a good indicator of its efficacy. This gives us a good idea about the re-engagement capabilities of the bot. This insight can be used to further finetune the but and make it par with the user requirements. Once completed the intended graph would look similar to the below sample graph



Retention Rate

Retention rate is the amount of user who come back within a brief period of time, this again is a good indicator of your chatbots efficiency A typical retention report would look like this



	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
September 9	58%	64%	38%	12%	96%	97%	48%
September 10	92%	123%	16%	32%	98%	85%	31%
September 11	67%	15%	19%	88%	88%	30%	114%
September 12	24%	35%	114%	115%	44%	32%	47%
September 13	8%	48%	132%	117%	13%	82%	84%
September 14	19%	117%	72%	8%	120%	1%	64%
September 15	10%	78%	52%	5%	6%	31%	123%

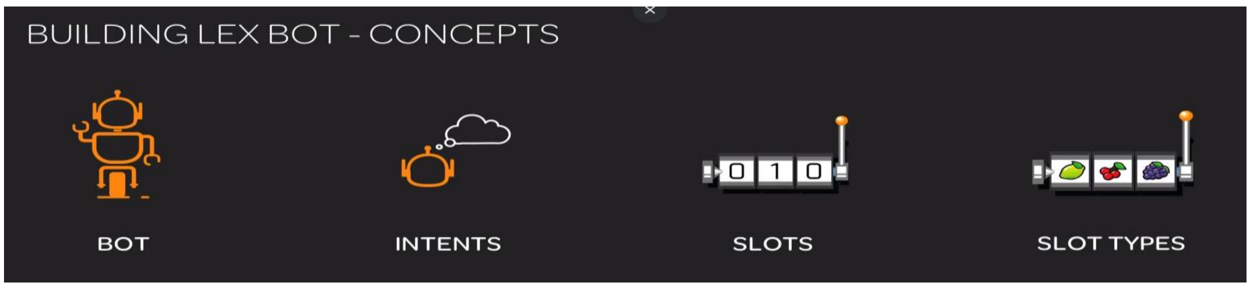
Number of People: 4700

0
25
50
75
100
125
150

AWS Lex

AWS has multiple services available, we are using AWS Lex to build our bot. AWS Lex is used for building conversational interfaces into any application using voice and text.
There are many options to build a chatbot, but we chose AWS Lex because of the easy and accurate integration with other services of AWS. Also, AWS has an option to create the serverless function for the validation and fulfilment process.
Friday is an Informational Bot, which would address the user queries related to movies.

Key Features:



1. **Intents**: The intents will perform the action based on the input given by the user. For example: In Friday there are multiple intents, best example is the Greeting intent.

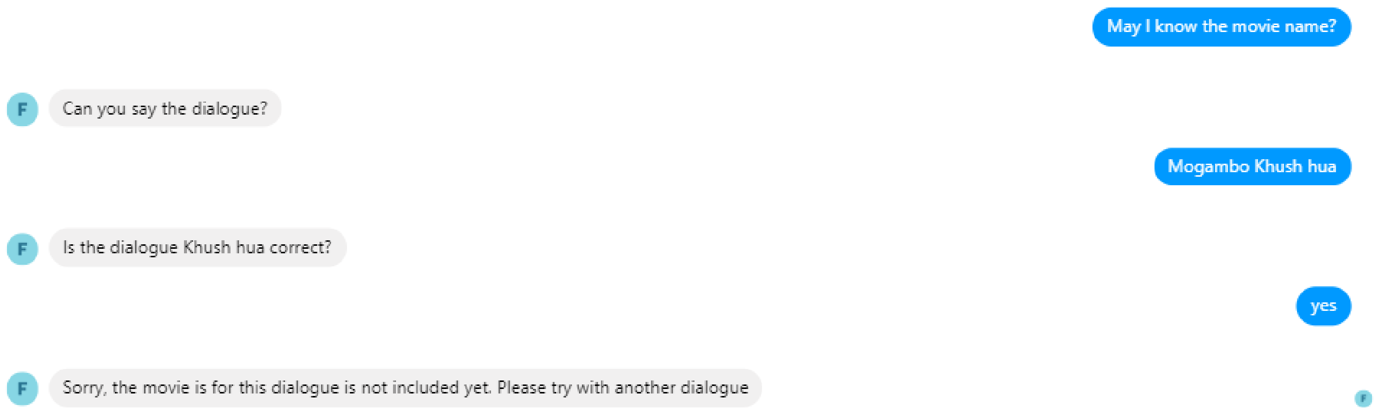
   When the user says Hi, Hello or Greetings, the Greeting intent is invoked and the bot responds back with the dialogues as shown in the screenshot.
   Other intents in Friday are movie Dialogue, movie review and goodbye. Each of these perform actions accordingly.

2. **Utterances**: The input provided by the user in text or voice format to invoke an intent is called the utterance. One Intent can have multiple utterances which can invoke that particular intent.
   For example, the utterance for Greeting intent is Hi, Hello or Greetings. Similarly, we have many utterances for the intents.

3. **Slots**: The input data that has to be fetched from the utterance is the slot. The slot value is then used to perform the next steps. We can use Amazon provided slots or we can define custom slots with the types allowed for each slot.

For example, Movie Dialogue is a slot in case of movie intent. Movie name is the slot for the movie review intent. The movie name for the movie review intent is either entered by the user or can be collected from the session attributes of the movie dialogue intent if the user requests for the movie review of the same movie as the movie dialogue.

4. **Prompts**: AWS Lex has two types of prompts: confirmation prompt and error handling prompt. The confirmation prompt validates the user input before performing the action, "Is the dialogue "movie dialogue" correct?" Is the confirmation prompt for Friday after the user enters the movie dialogue?
"Sorry, the movie is for this dialogue is not included yet. Please try with another dialogue" is the error handling prompt when the movie dialogue is not included in the database.


May I know the movie name?

Can you say the dialogue?

Mogambo Khush hua

Is the dialogue Khush hua correct?

yes

Sorry, the movie is for this dialogue is not included yet. Please try with another dialogue

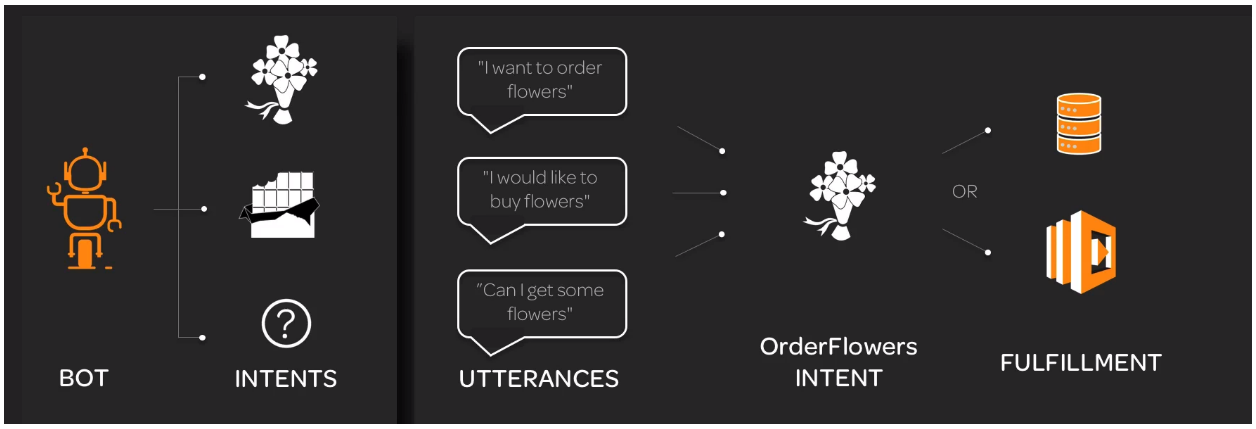
5. **Validation Function**: Validation can be performed on the user entered inputs based on the logic provided in the Validation Lambda function. The confirmation logic performed by the user is also handled by the validation function.
For example, when the user enters the movie dialogue, the bot confirms whether the dialogue the bot has saved in the movie dialogue slot is correct.
The user has the option to agree or disagree. Then once the user validates next steps are performed as per the business logic.

6. **Fulfilment Function**: Fulfilment has the backend function that has to be performed based on the user intent.
For example, If the user request for the movie name based on the dialogue then the chatbot has to search the dialogue in the database, if found the corresponding movie has to be displayed back.

7. **Integration of chatbot**: AWS Lex provides to integrate various messaging platforms like Facebook Messenger, Slack, Kik, Twilio. Chatbot can also be integrated with mobile

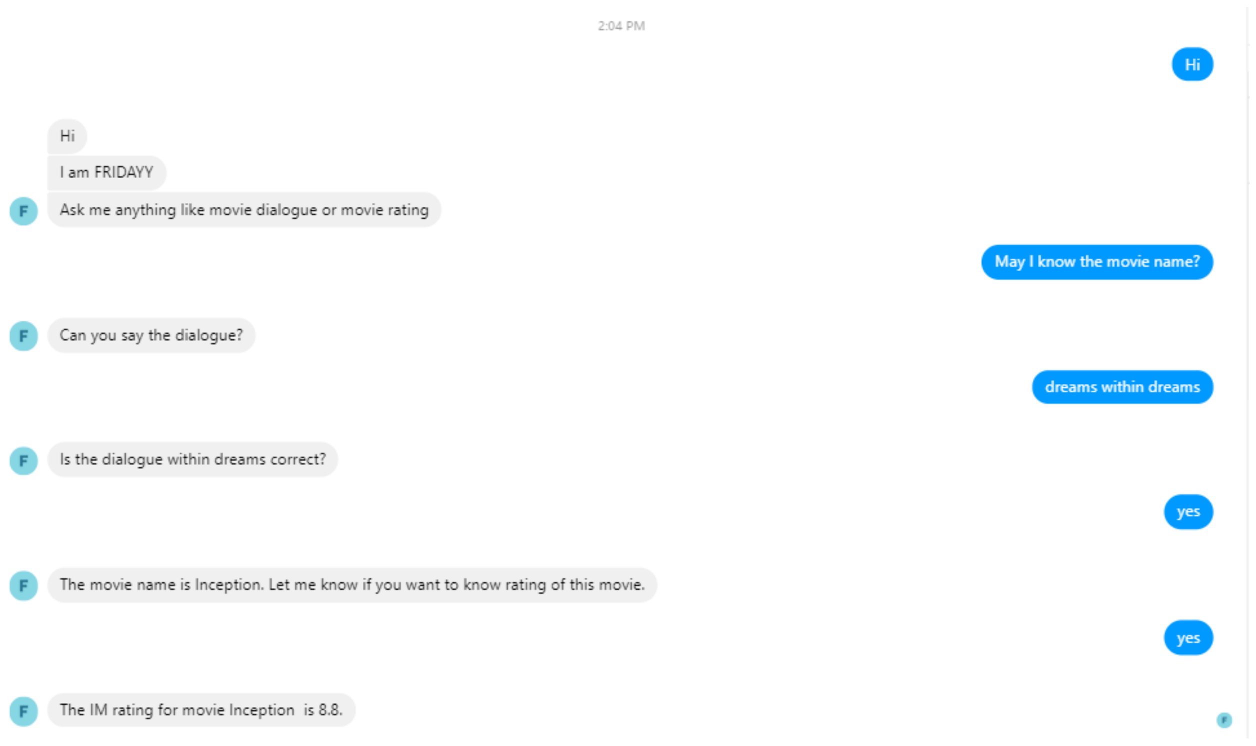
applications and websites. We have chosen Facebook messenger as our integration platform since it is the highly used social media platform across the globe.

Working of Lex:



The diagram above shows the working of chatbot. Here, when the utterances are entered by the user, the corresponding intent is invoked. Then the validation or fulfillment functions are invoked for that particular intent.

Below is the screenshot of Friday chatbot from Facebook messenger.


2:04 PM

Hi

Hi

I am FRIDAYY

Ask me anything like movie dialogue or movie rating

May I know the movie name?

Can you say the dialogue?

dreams within dreams

Is the dialogue within dreams correct?

yes

The movie name is Inception. Let me know if you want to know rating of this movie.

yes

The IM rating for movie Inception is 8.8.

Recommendation Engine

A recommender system refers to a system that is capable of predicting the future preference of a set of items for a user and recommend the top items. One key reason why we need a recommender system is that people have too much options to use from due to the prevalence of Internet.

Although the amount of available information increased, a new problem arose as people had a hard time selecting the items they actually want to see. This is where the recommender system comes in.

Recommender system enables to narrow down the dimensions of the information and suggest the relevant information to the end user.

Types of Recommendation Engines

1. **Content-Based**

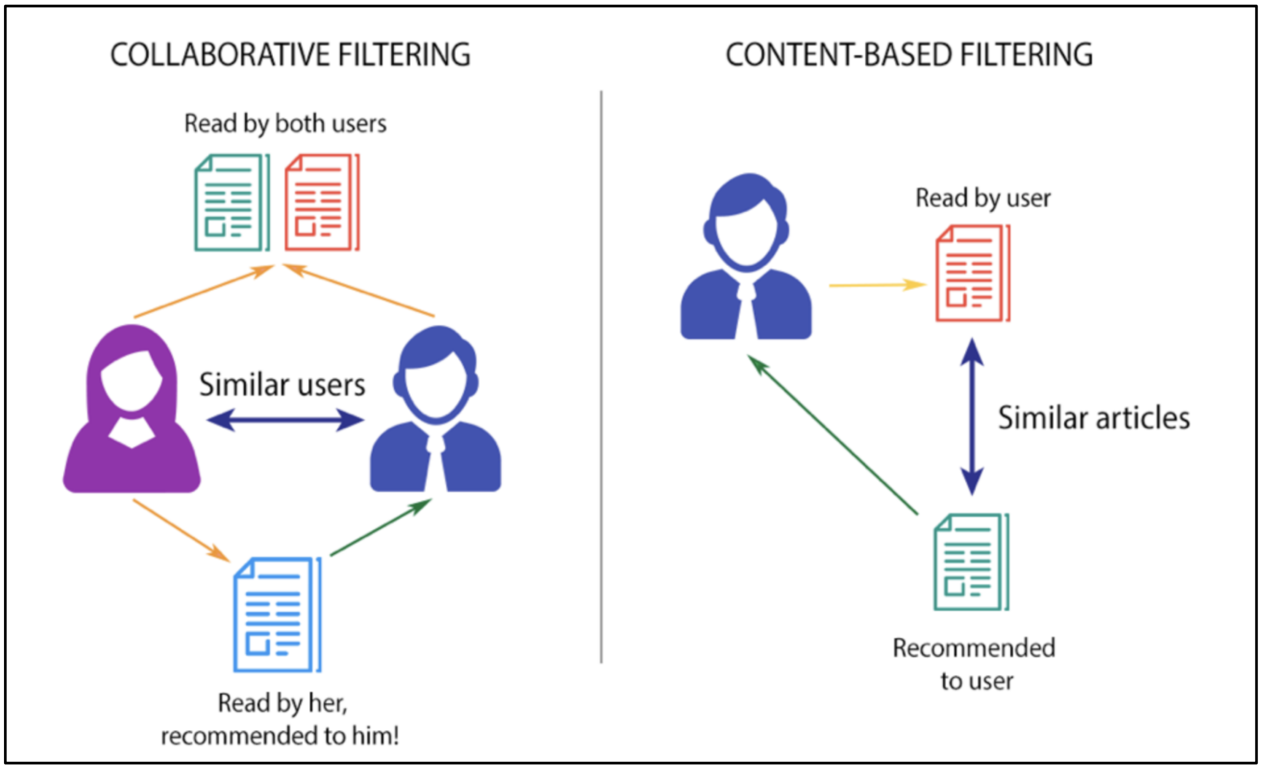
The Content-Based Recommender relies on the similarity of the items being recommended. The basic idea is that if you like an item, then you will also like a “similar” item. It generally works well when it's easy to determine the context/properties of each item.

2. Collaborative Filtering

The Collaborative Filtering Recommender is entirely based on the past behavior and not on the context. More specifically, it is based on the similarity in preferences, tastes and choices of two users. It analyses how similar the tastes of one user is to another and makes recommendations on the basis of that.

For instance, if user A likes movies 1, 2, 3 and user B likes movies 2,3,4, then they have similar interests and A should like movie 4 and B should like movie This makes it one of the most commonly used algorithms as it is not dependent on any additional information.

In this project we have applied collaborative filtering using different techniques and optimized our solutions using Singular vector dimension.



Collaborative Filtering Recommendation Model

The content-based engine suffers from some severe limitations. It is only capable of suggesting movies which are close to a certain movie. That is, it is not capable of capturing tastes and providing recommendations across genres. Also, it doesn't capture the personal tastes and biases of a user. Anyone querying our engine for recommendations based on a movie will receive the same recommendations for that movie, regardless of who the user is.

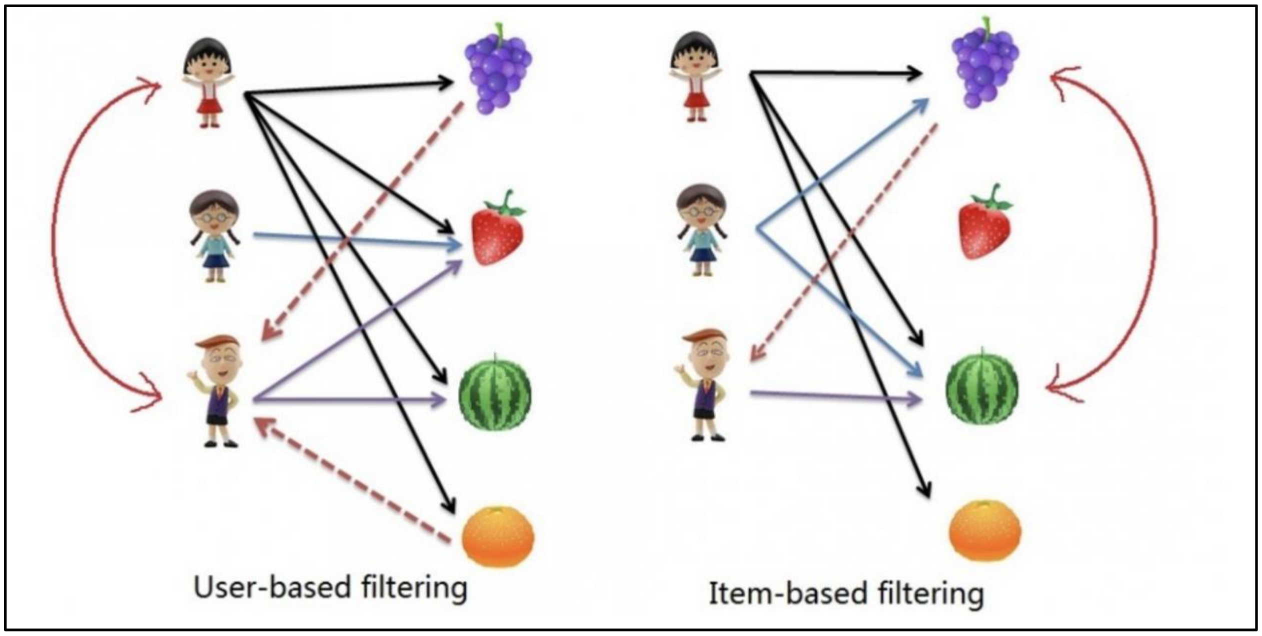
Therefore, the technique is based on the idea that users similar to a me can be used to predict how much I will like a particular product or service those users have used/experienced, but I have not.

Theory

There are 2 main types of memory-based collaborative filtering algorithms:

1. **User-User Collaborative Filtering**: Here we find look alike users based on similarity and recommend movies which first user's look-alike has chosen in past. This algorithm is very effective but takes a lot of time and resources. It requires to compute every user pair information which takes time. Therefore, for big base platforms, this algorithm is hard to implement without a very strong parallelizable system.

2. **Item-Item Collaborative Filtering**: It is quite similar to previous algorithm, but instead of finding user's look-alike, we try finding movie's look-alike. Once we have movie's look-alike matrix, we can easily recommend alike movies to user who have rated any movie from the dataset. This algorithm is far less resource consuming than user-user collaborative filtering. Hence, for a new user, the algorithm takes far lesser time than user-user collaborate as we don't need all similarity scores between users. And with fixed number of movies, movie-movie look alike matrix is fixed over time.



In either scenario, we need build a similarity matrix.

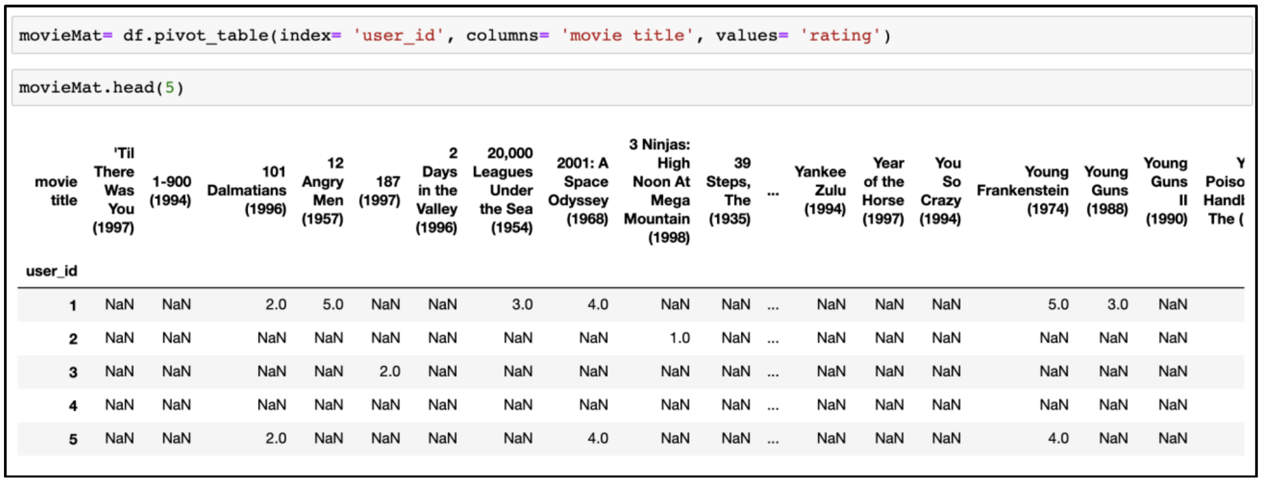
For user-user collaborative filtering, the **user-user similarity matrix** will consist of some distance metrics that measure the similarity between any two pairs of users.

And for the **item-item similarity matrix** will measure the similarity between any two pairs of movies.

There are 2 distance similarity metrics that we are using in our solution.

1. **Cosine Similarity**:
   - Similarity is the cosine of the angle between the 2 vectors of the movie' s vectors of A and B
   - Closer the vectors, smaller will be the angle and larger the cosine
2. **Pearson Similarity**:
   - Similarity is the Pearson coefficient between the two vectors.

CREATING THE MATRIX:

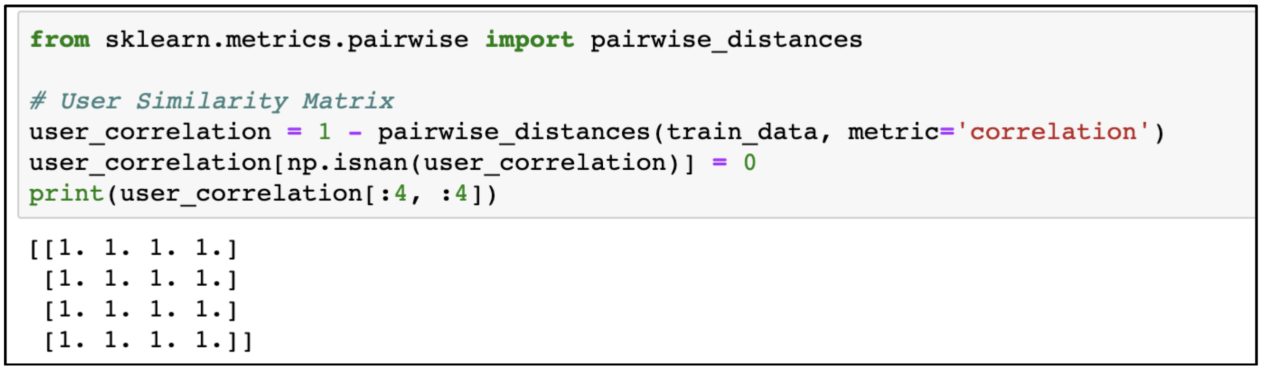


```
movieMat= df.pivot_table(index= 'user_id', columns= 'movie title', values= 'rating')
```

```
movieMat.head(5)
```

movie title	'Til There Was You (1997)	1-900 (1994)	101 Dalmatians (1996)	12 Angry Men (1957)	187 (1997)	2 Days in the Valley (1996)	20,000 Leagues Under the Sea (1954)	2001: A Space Odyssey (1968)	3 Ninjas: High Noon At Mega Mountain (1998)	39 Steps, The (1935)	...	Yankee Zulu (1994)	Year of the Horse (1997)	You So Crazy (1994)	Young Frankenstein (1974)	Young Guns (1988)	Young Guns II (1990)	Y Poiso Handl The (
user_id																		
1	NaN	NaN	2.0	5.0	NaN	NaN	3.0	4.0	NaN	NaN	...	NaN	NaN	NaN	5.0	3.0	NaN	
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
3	NaN	NaN	NaN	NaN	2.0	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
5	NaN	NaN	2.0	NaN	NaN	NaN	NaN	4.0	NaN	NaN	...	NaN	NaN	NaN	4.0	NaN	NaN	

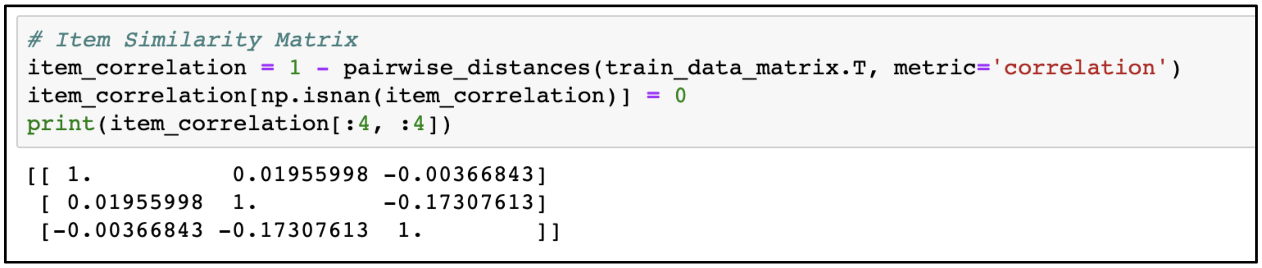
USER- USER BASED SIMILARITY USING CORRELATION DISTANCE METRICS:



```
from sklearn.metrics.pairwise import pairwise_distances

# User Similarity Matrix
user_correlation = 1 - pairwise_distances(train_data, metric='correlation')
user_correlation[np.isnan(user_correlation)] = 0
print(user_correlation[:4, :4])
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

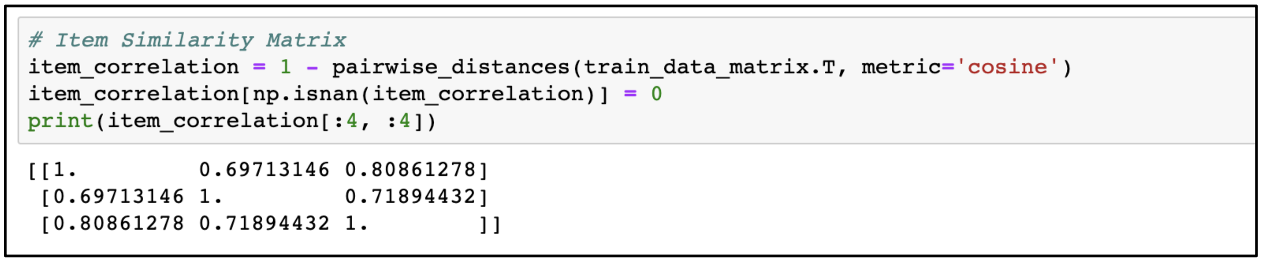
ITEM- ITEM BASED SIMILARITY USING Correlation METRICS:



```
# Item Similarity Matrix
item_correlation = 1 - pairwise_distances(train_data_matrix.T, metric='correlation')
item_correlation[np.isnan(item_correlation)] = 0
print(item_correlation[:4, :4])
```

```
[[ 1.          0.01955998 -0.00366843]
 [ 0.01955998  1.         -0.17307613]
 [-0.00366843 -0.17307613  1.        ]]
```

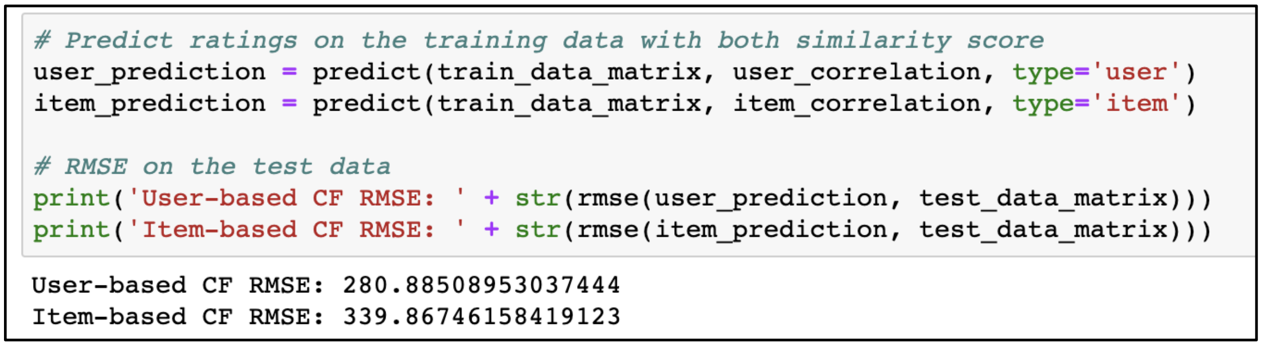
ITEM- ITEM BASED SIMILARITY USING cosine METRICS:



```
# Item Similarity Matrix
item_correlation = 1 - pairwise_distances(train_data_matrix.T, metric='cosine')
item_correlation[np.isnan(item_correlation)] = 0
print(item_correlation[:4, :4])
```

```
[[1.         0.69713146 0.80861278]
 [0.69713146 1.         0.71894432]
 [0.80861278 0.71894432 1.        ]]
```

BELOW EVALUATION SHOWS THE RMSE FOR COLLABORATIVE FILTERING USING



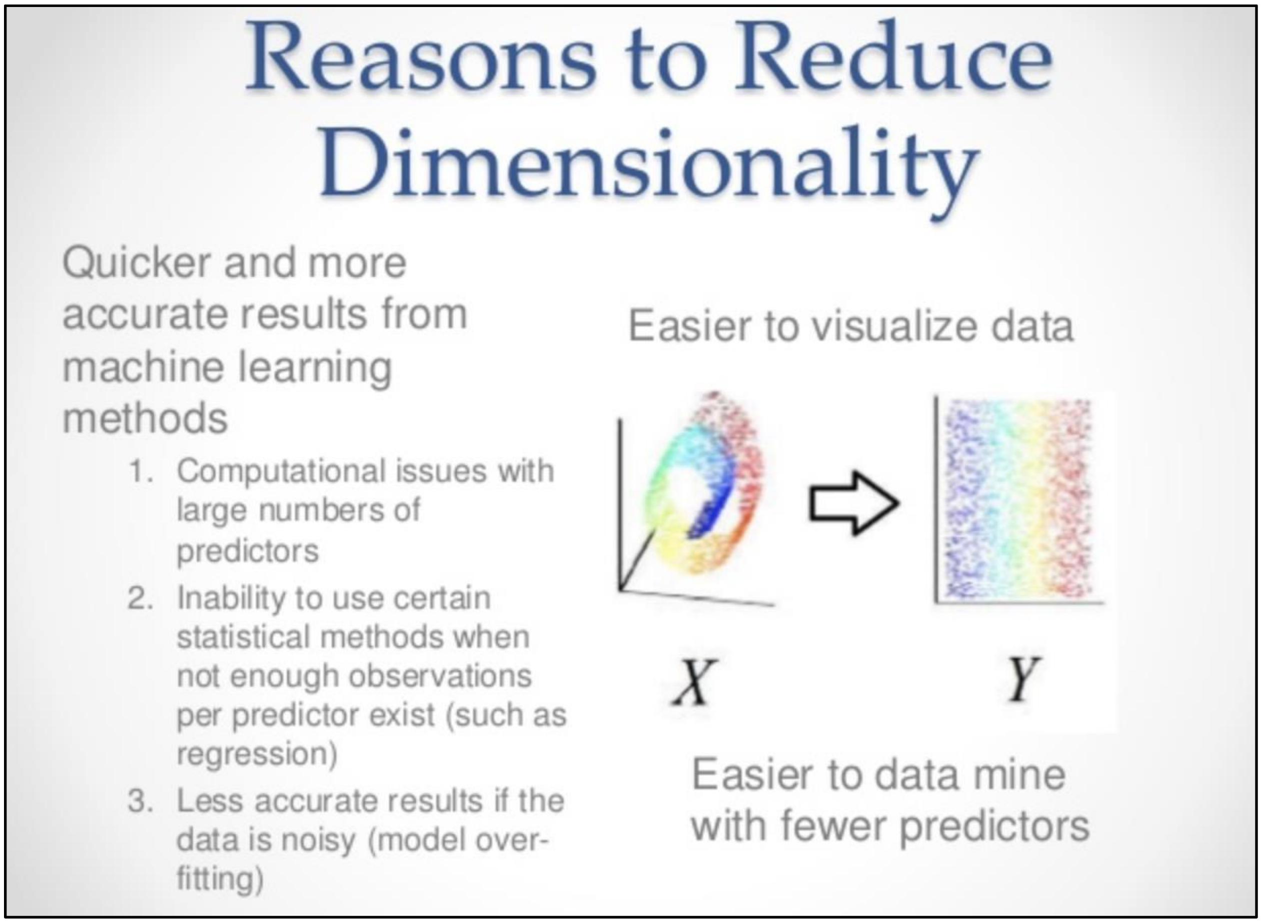
```
# Predict ratings on the training data with both similarity score
user_prediction = predict(train_data_matrix, user_correlation, type='user')
item_prediction = predict(train_data_matrix, item_correlation, type='item')

# RMSE on the test data
print('User-based CF RMSE: ' + str(rmse(user_prediction, test_data_matrix)))
print('Item-based CF RMSE: ' + str(rmse(item_prediction, test_data_matrix)))
```

```
User-based CF RMSE: 280.88508953037444
Item-based CF RMSE: 339.86746158419123
```

There are two major issues while computing the distance relationships between items or users have these two major issues:

1. It doesn't scale particularly well to massive datasets, especially for real-time recommendations based on user behavior similarities - which takes a lot of computations.
2. Ratings matrices may be overfitting to noisy representations of user tastes and preferences. When we use distance based "neighborhood" approaches on raw data, we match to sparse low-level details that we assume represent the user's preference vector instead of the vector itself.

Thus, we have optimized the solution using SVD Dimensionality Reduction technique to derive the tastes and preferences from the raw data, this technique is also known as doing low-rank matrix factorization.



Reasons to Reduce Dimensionality

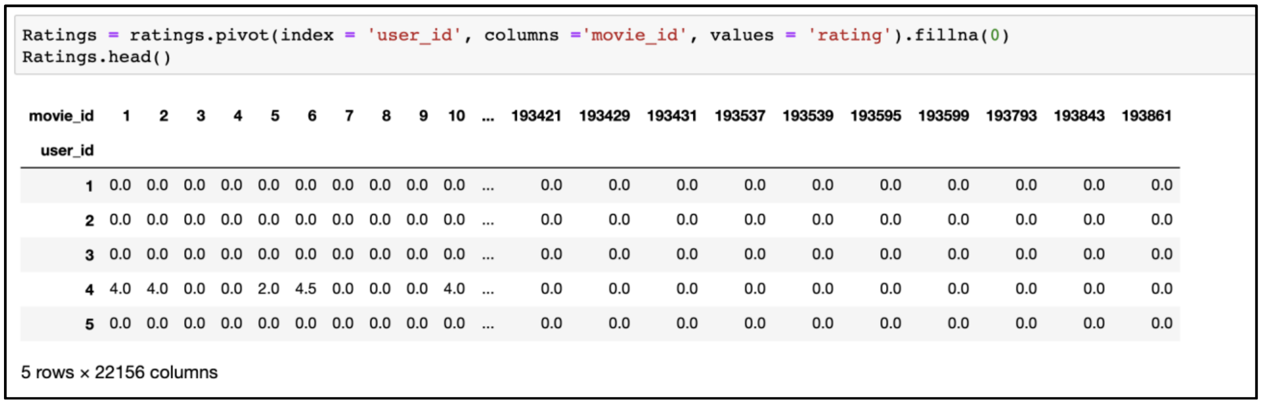
Quicker and more accurate results from machine learning methods

1. Computational issues with large numbers of predictors
2. Inability to use certain statistical methods when not enough observations per predictor exist (such as regression)
3. Less accurate results if the data is noisy (model over-fitting)

Easier to visualize data

Easier to data mine with fewer predictors

THIS IS HOW OUR UTILITY MATRIX LOOKS LIKE. WE CAN CLEARLY NOTICE THE SPARSITY OF THIS MATRIX.

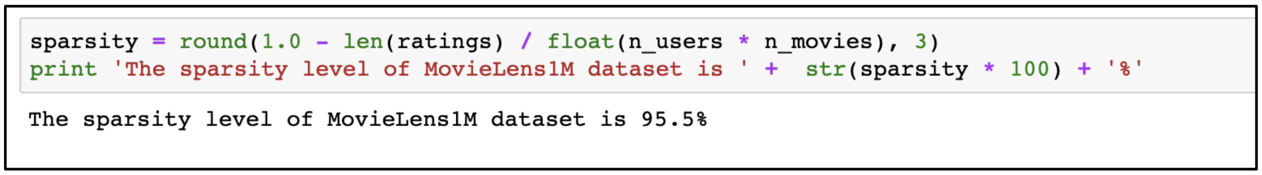


```
Ratings = ratings.pivot(index = 'user_id', columns ='movie_id', values = 'rating').fillna(0)
Ratings.head()
```

movie_id	1	2	3	4	5	6	7	8	9	10	...	193421	193429	193431	193537	193539	193595	193599	193793	193843	193861
user_id																					
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	4.0	4.0	0.0	0.0	2.0	4.5	0.0	0.0	0.0	4.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 22156 columns

WHILE CHECKING THE SPARSITY OF OUR MATRIX, WE CAN SEE THAT OUR MATRIX IS ABOUT 96 PERCENT SPARSED.



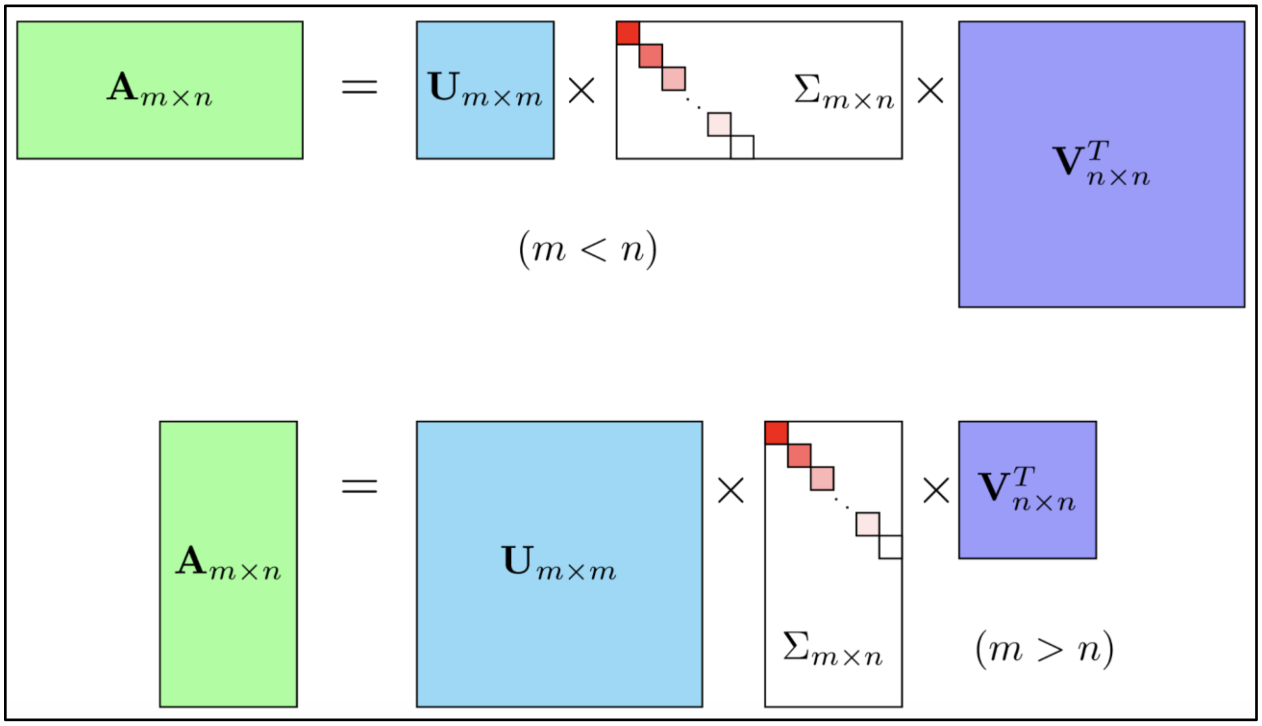
```
sparsity = round(1.0 - len(ratings) / float(n_users * n_movies), 3)
print 'The sparsity level of MovieLens1M dataset is ' +  str(sparsity * 100) + '%'
```

```
The sparsity level of MovieLens1M dataset is 95.5%
```

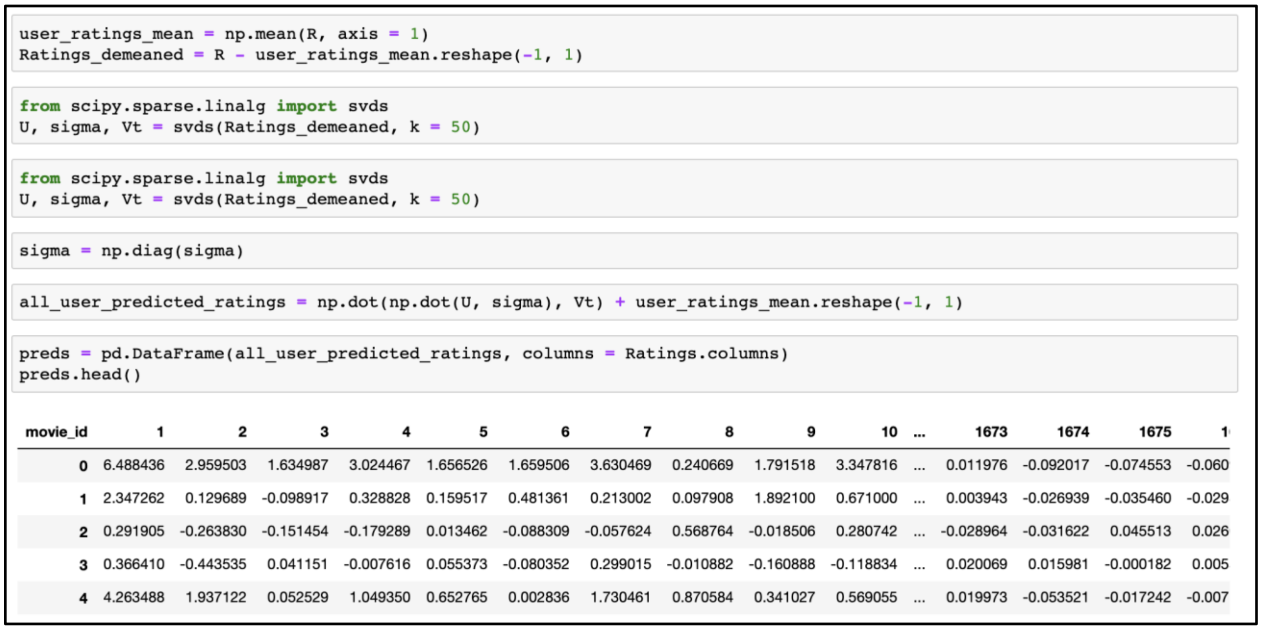
SVD CONCEPT

The essence of SVD is that it decomposes a matrix of any shape into a product of 3 matrices with nice mathematical properties: $A=USVT$A=USVT.

By lucid analogy, a number can decompose into 3 numbers to always have the smallest prime in the middle.



DECOMPOSING THE SPARSE MATRIX:



```
user_ratings_mean = np.mean(R, axis = 1)
Ratings_demeaned = R - user_ratings_mean.reshape(-1, 1)
```

```
from scipy.sparse.linalg import svds
U, sigma, Vt = svds(Ratings_demeaned, k = 50)
```

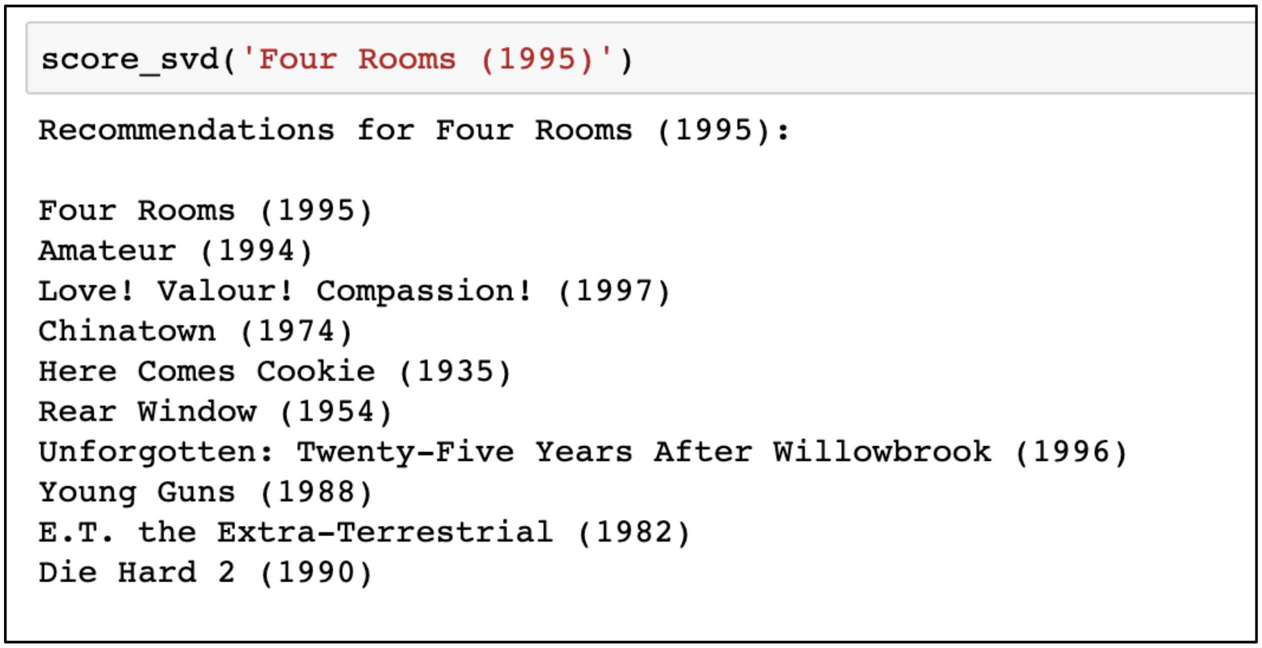
```
from scipy.sparse.linalg import svds
U, sigma, Vt = svds(Ratings_demeaned, k = 50)
```

```
sigma = np.diag(sigma)
```

```
all_user_predicted_ratings = np.dot(np.dot(U, sigma), Vt) + user_ratings_mean.reshape(-1, 1)
```

```
preds = pd.DataFrame(all_user_predicted_ratings, columns = Ratings.columns)
preds.head()
```

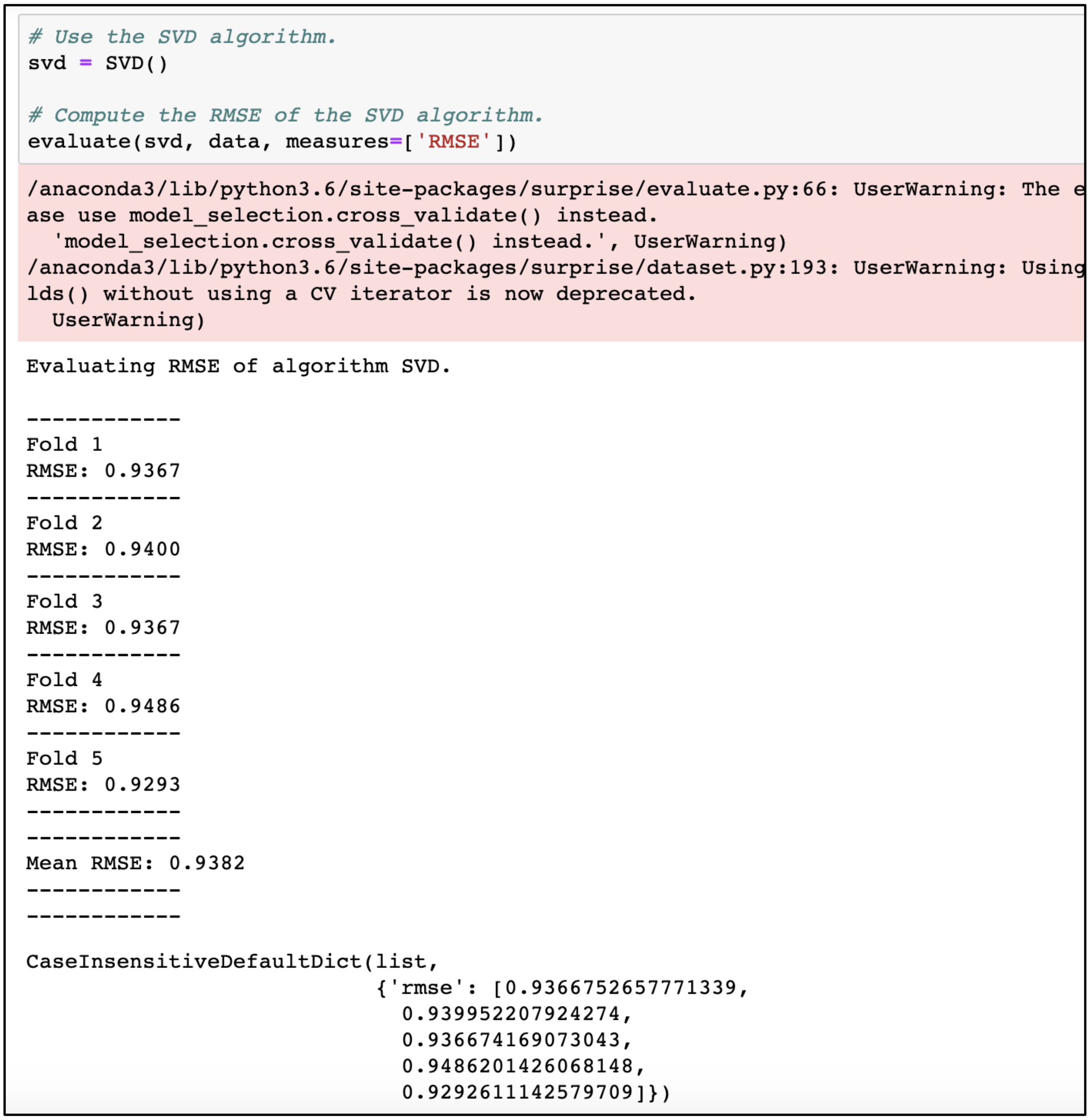
movie_id	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	1
0	6.488436	2.959503	1.634987	3.024467	1.656526	1.659506	3.630469	0.240669	1.791518	3.347816	...	0.011976	-0.092017	-0.074553	-0.060
1	2.347262	0.129689	-0.098917	0.328828	0.159517	0.481361	0.213002	0.097908	1.892100	0.671000	...	0.003943	-0.026939	-0.035460	-0.029
2	0.291905	-0.263830	-0.151454	-0.179289	0.013462	-0.088309	-0.057624	0.568764	-0.018506	0.280742	...	-0.028964	-0.031622	0.045513	0.026
3	0.366410	-0.443535	0.041151	-0.007616	0.055373	-0.080352	0.299015	-0.010882	-0.160888	-0.118834	...	0.020069	0.015981	-0.000182	0.005
4	4.263488	1.937122	0.052529	1.049350	0.652765	0.002836	1.730461	0.870584	0.341027	0.569055	...	0.019973	-0.053521	-0.017242	-0.007



```
score_svd('Four Rooms (1995)')
```

```
Recommendations for Four Rooms (1995):

Four Rooms (1995)
Amateur (1994)
Love! Valour! Compassion! (1997)
Chinatown (1974)
Here Comes Cookie (1935)
Rear Window (1954)
Unforgotten: Twenty-Five Years After Willowbrook (1996)
Young Guns (1988)
E.T. the Extra-Terrestrial (1982)
Die Hard 2 (1990)
```

EVALUATION OF SVD ALGORITHM:



```
# Use the SVD algorithm.
svd = SVD()

# Compute the RMSE of the SVD algorithm.
evaluate(svd, data, measures=['RMSE'])
```

```
/anaconda3/lib/python3.6/site-packages/surprise/evaluate.py:66: UserWarning: The e
ase use model_selection.cross_validate() instead.
  'model_selection.cross_validate() instead.', UserWarning)
/anaconda3/lib/python3.6/site-packages/surprise/dataset.py:193: UserWarning: Using
lds() without using a CV iterator is now deprecated.
  UserWarning)
```

```
Evaluating RMSE of algorithm SVD.

------------
Fold 1
RMSE: 0.9367
------------
Fold 2
RMSE: 0.9400
------------
Fold 3
RMSE: 0.9367
------------
Fold 4
RMSE: 0.9486
------------
Fold 5
RMSE: 0.9293
------------
------------
Mean RMSE: 0.9382
------------
------------

CaseInsensitiveDefaultDict(list,
                           {'rmse': [0.9366752657771339,
                             0.939952207924274,
                             0.936674169073043,
                             0.9486201426068148,
                             0.9292611142579709]})
```

Challenges Faced :

- Data Storage
  - There were multiple challenges that we faced during the storage of data, because of the following reasons

- We had two million records that had to be stored in a database, which had to be searched and retrieved within milliseconds
- The movie dialogue dataset had all the detailed scripts of more than 2000 movies and the search parameter is a string (String of Dialogue from a movies)
- We have tried multiple combination of storage and each had their own set of problems which are discussed below in the paper
    - S3 and Lambda Function
        - Initially we thought this to be the most optimized way of data storage that would suit our requirement. Cause we have to find the name of the movie based on the dialogue as an input. The absence of any key, as the part of the input made to very difficult for us to use relational or unstructured databases. This forced us to think for other storage options, like S3 and use lambda function to trigger bash scripts within S3. While this was a tangible option, but the time consumed by the lambda function to trigger a bash script within S3 was very high which increased the search period and inefficient
    - EC2 , EBS and lambda
        - As we wanted to trigger bash scripts using lambda function and the performance with S3 was not great, we thought of hosting an UNIX server using EC2 and store the data using EBS and instance stores
        - We tried hosting an API using, flask and triggering the bash script from the lambda function
        - We were not able to pass the search string as part of the input parameter using the api call and extracting the name of movie hence became difficult
    - Dynamo DB and DAX (Dynamo DB accelerator)
        - Storage and retrieval of unstructured data in Dynamo DB, primarily depends on indexes and indexes are basically a combination of sort and partition key which helps to optimized data retrieval
        - For efficient use of the of indexes the input should be a key and any search value, our problem was that we did not had any key that could be passed along with the Search string
        - While DB allows keyless search, this is an inefficient process, because keyless search would require performing a scan query, where the entire dataset has to loaded into the memory and the over all read capacity units consumed is 1.5 times the query operation
    - Relational Database services (RDS)

        - The data we had in had was mostly unstructured and lack of any key during the search forced us to look for storage options beyond RDS
- Retrieving data from Dynamo DB using lambda
    - As we performed a keyless scan operation in the Dynamo DB, this took some time to complete and we had to increase the default 3 second time out period of the lambda function also the default memory allocated to the lambda function had to be increased to accommodate the operation
- Creating Sagemaker Endpoint
    - Although Sagemaker is a managed AWS there are no direct way to consume API, and in order to consume you either have to host an endpoint or package your code as a function, both of this further complicated thing for us

Experience & Key Learnings

The overall experience and the learning curve for this entire project was great. From the outset we were forced to think out of the box, because until now we were used to completing projects, The concept of Product was new to us and forced us to think out of the box, also the requirement of coming up with something that would be addressing a particular business case further complicated things.

The learning curve for us during the entire process was very steep because of the following reasons

- Conceptualizing the entire product from the scratch demanded deeper research and thinking out of the box
- Architecting the product and finalizing on which components to use
- Hosting a Serverless microservices based application had its own complications
- We realized that while developing individual units of the application was easy, integrating them was one of the most difficult task
- Building a recommendation engine from the scratch was also a demanding task

Future Enhancements:

1. Including scripts and datasets for more movies, including movies from different languages.
2. Integration of Voice inputs.
3. Integration of recommendation engine.
4. Use of elastic search cluster to optimize search operations