Neural Network-Based Movie Recommendation System

A study of Collaborative and Context-Based Filtering Approaches

1. Motivation

1.1 The Information Age

The inspiration for this project comes from the current era's life trends. Technology has brought about a change in power dynamics at the systemic level. Better access to more products has made brand loyalty less secure. This has forced firms to rethink their marketing strategies. Having the right strategy would help them build customer loyalty and long-term relationships with their customers, besides making them profitable and more efficient. It was soon clear that acknowledging and understanding customer behavior and personalities lays the foundation for such a strategy.

1.2 The Origin of Behavioral Economics

Though firms had tried to build such a strategy using Psychology theories and Neo-Classical Economics, they were not as successful as they had expected. It was the origin of Behavioral Economics that shaped the future of the contemporary, successful, and most customer-centric firms in various industries. Unlike Neo-classical Economics which assumes that most people have well-defined preferences and make well-informed, self-interested decisions based on those preferences, Behavioral Economics tries to explain why people behave the way they do in real life.

1.3 The Era of Customer Personality Analysis

Richard Thaler, the father of Behavioral Economics, soon showed that personality traits act as predictors of shopping motivators and behaviors. Since then, successful firms have been using the *Segmentation-Targeting-Positioning (STP) Framework* based on customer personality analysis to excel in cost, revenue, and customer satisfaction perspectives. The success of the firm, thus, heavily relies on its ability to build robust recommendation systems using those segments.

In this project, I aim to perform the same experiment of segmenting customers and building a recommendation system using two possible methodologies and compare their results. The project, thus, lies at the intersection of Behavioral Economics and Deep Learning.

2. The Project Idea

This project aims to build a decently-accurate deep learning model, trained for recommending movies to customers. The project thus attempts to perform exploratory data analysis and build neural networks based on collaborative filtering and content-based filtering which are well-suited for the context. The study objective is to explore different techniques, examine the outcomes, and draw some conclusions.

The dataset, published on Kaggle, features over 20 Million ratings and free-text tagging activities from MovieLens. It contains 20000263 ratings and 465564 tag applications across 27278 movies. These data were created by 138493 users between January 09, 1995, and March 31, 2015. Users were selected at random for inclusion. All selected users had rated at least 20 movies. I believe this will ensure that the model is not biased.

2.1 Project Logistics

Dataset: MovieLens 20M
Dataset Structure: The data are contained in six files which can be merged according to usage
Dataset Dimensions:

- 20000263 ratings,
- 465564 tag applications across 27278 movies138493 users
-

But due to resource crunch in terms of computing available to me on Google Colab and the time it takes to process more epochs, I have only used a subset of the dataset i.e. I have taken 1.4M records, which when augmented amounts to 7M data points
Packages Used:

- Data Handler: Pandas, NumPy
- Models: Pytorch, PyTorch-lightning

Platform: Google Colab

2.2 Project Plan

- Exploratory Data Analysis
    - Involves combining and preprocessing data from six different files according to the model's needs. For example, Content-Based Filtering requires the genres of the movies but Collaborative Filtering does not
    - Perform Data Visualization
        - Verify the existence of multi-collinearity
        - Inspect correlation using heatmaps
        - Univariate analysis using boxplots
- Model Building
    - Collaborative Filtering
    - Content-Based Filtering
- Assessment and Evaluation
    - Metrics
        - Recall and Precision
        - Normalized Discounted Cumulative Gain (NDCG)
        - Diversity
        - Coverage
        - Novelty
- Conclusions
- Report

2.2 The Impact of the Project

Using behavioral analytics, firms can attain access to customer attitudes and personalities which in turn can help in transforming services, sales, collections, and care management. But how does that work? Though firms build their products based on user research, how the product is marketed, advertised, and positioned in the market is of paramount importance too. This is where Richard Thaler's *Nudge Theory* comes into play.

> *"Nudge Theory is based upon the idea that by shaping the environment, also known as the choice architecture, one can influence the likelihood that one option is chosen over another by individuals"*

By taking into account many of the factors that influence individuals and shaping the environment to nudge customers to purchase, firms can leverage indirect methods to influence their customers' choices. But, firms can only get average outcomes if they group all of their customers into one bucket. Consequently, Nudge theory only when combined with data of the past like action histories, demographics, etc. which help in creating theories of customers can lay the foundations for a roadmap to success and differentiation for the firm's products.

Many multinational organizations have personalized their products through high-performing recommendation systems. This project aims to build hypotheses and conclusions about the performance of two major techniques of recommendation systems using the dataset at hand.

3. Literature Survey

Brand diversity has been growing exponentially. Equally fierce is the rivalry in each industry as firms toil to retain their customers and stay profitable. In such an environment, there has been a lot of research in the field of personalization using statistics and deep learning. Based on the numerous research papers I have referred to, it can be understood that Collaborative filtering is among the most widely applied approaches in recommender systems. Collaborative filtering predicts what items a user will prefer by discovering and exploiting the similarity patterns across users. Another major paradigm in the spotlight is Content-Based filtering which predicts the items a user will prefer using the similarity between items.

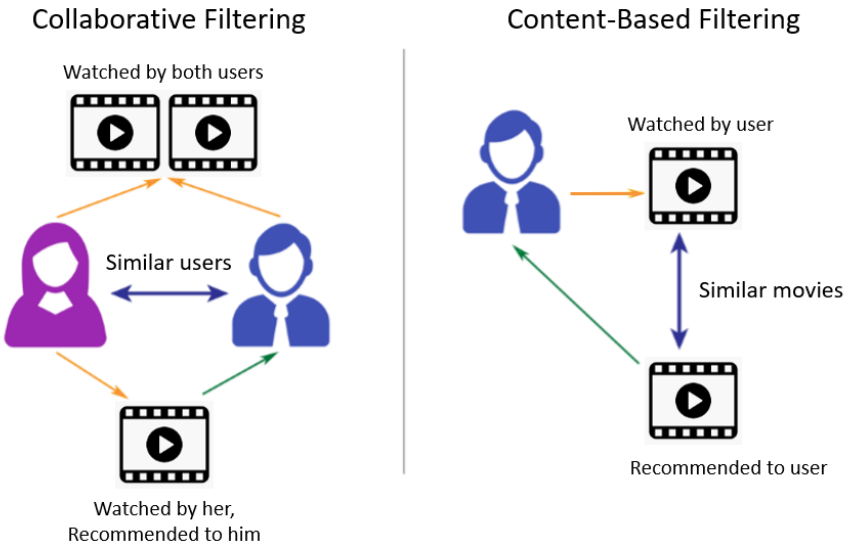


Image source: KDnuggets

3. 1 Terminology

3.1.1 Collaborative Filtering

Inputs from various users are used by collaborative systems, i.e, it is based on user-group behavior. These models evaluate these inputs in numerous ways to calculate similarity measures between the users. They build models based on the user's prior behavior which includes activity, ratings, and favorites. The products liked by similar users are then recommended to the user in question. There are two types of collaborative filtering models

- **Memory-Based Models**

  Memory-based approaches continuously analyze user data to make recommendations. As they utilize the user ratings, they gradually improve in accuracy over time. They are domain-independent and do not require content analysis [3]. They are further categorized as follows

  - Item-Based: “Users who liked this item also liked…”
  - User-Based: “Users who are similar to you also liked…”

- **Model-Based Models**

  Model-based approaches develop a model of a user’s behavior and then use certain parameters to predict future behavior [3]. They use machine learning models to model the implicit feedback expected by the recommendation system to predict future activity.

3.1.2 Content-Based Filtering

Content-based models, on the other hand, rely on product features to suggest recommendations. Meta-data of the product is used to calculate similarity and hence these modes are domain-dependent. They make use of a user’s particular interests and attempt to match a user’s profile to the attributes possessed by the various content objects to be recommended [3]

3.1.3 Hybrid Models

Hybrid systems combine collaborative and content-based filtering systems, to optimize the recommender systems, and reduce the drawbacks present in each of the two methods. These models rely on different methods to combine the results of two models like weighted combiner, switching, feature augmentation, etc [6]

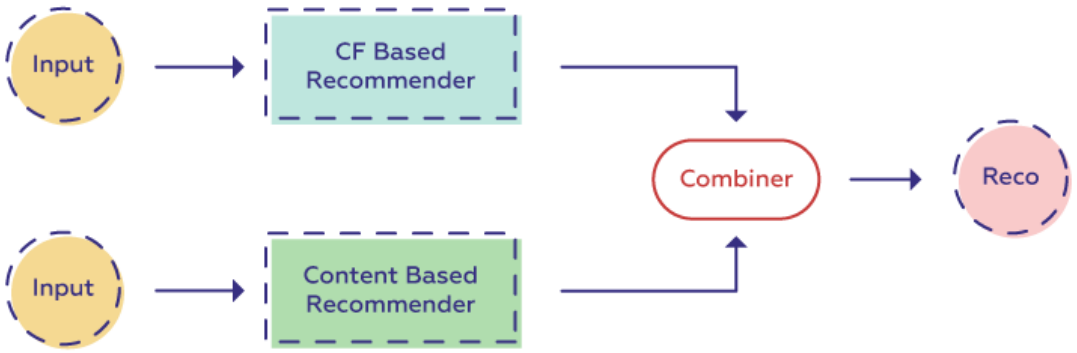


Image Source: Sciforce | Medium

3.2 Related Work

Recommender systems, which have been widely implemented by many online businesses, such as e-commerce, online news, and social networking sites, play a crucial role in reducing information overload in the age of information explosion. They help attain a win-win situation for customers and firms alike as customers get better recommendations and explore new products while firms build customer loyalty and increase sales and revenue.

A customized recommender system's secret recipe is in modeling consumers' preferences for products based on their prior interactions (such as ratings and clicks). The reason it is a secret recipe is that

there are multiple approaches to building one. There are collaborative models, content-based models, and hybrid models. The first-generation recommendation systems were purely associative rule-based, i.e knowledge-based systems. The second generation of recommendation systems brought in personalization using statistical methods like matrix factorization. The current generation of recommendation systems, however, are way more advanced and use implicit feedback from the users instead of relying on explicit feedback as in the case of previous generations. The third-generation models are being built using deep learning models which can capture user behavior and personality more accurately.

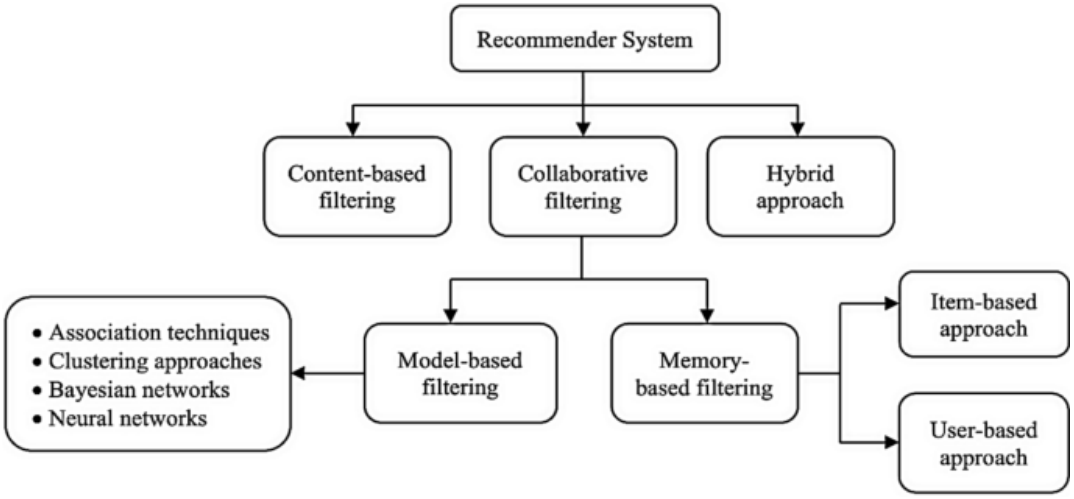
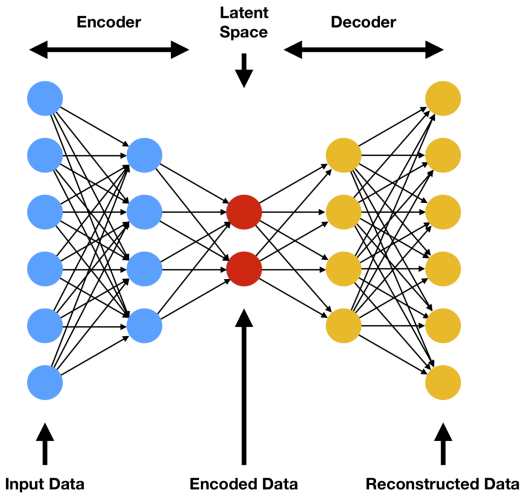


Image Source: ScienceDirect

In [1], extensive research has been carried out on model-based collaborative filtering. The authors proposed a generic framework for neural network-based models to be used in model-based filtering. The researchers point out that much research has been dedicated to improvising the Matrix Factorization approach such as integrating with neighborhood-based models, probabilistic functions, combining with content-based topic models, etc. But in essence, traditional techniques like factorization machines and matrix factorization are linear models which limits their modeling capability. Since neural networks have been well-known for their ability to model complex non-linear relations, they propose the NCF framework.

The major drawback of Matrix Factorization models is the drop in performance when sparsity steps in. In other words, as the content library grows i.e., the number of items increases, the percentage of ratings of the items decreases, making the rating matrix sparse. As users generally interact with a very small subset of content, matrix factorization was then used in combination with partition algorithms to reduce the search space for better performance. Later on, Neural networks were introduced to predict these user ratings for unseen items with greater accuracy when compared to the linear models of the above-mentioned models. In such pursuit, the researchers have proposed the use of variational autoencoders[2]. The work achieves state-of-the-art performance using multinomial likelihood instead of the more common gaussian and logistic likelihood.



Variational Autoencoder | Image Source: Compthree

For analyzing the temporal dynamics of user activity and item evolution, modeling sequential signals is an essential subject. Since deep neural networks are a good fit for such tasks, further research has been conducted in this field. Major developments have been next-basket prediction and session-based recommendations. Session-based systems have surfaced recently and have caught everyone's attention for bridging the research gap which ignores the dynamic nature of user preferences in the real world. For a current session of a user to be predicted (called target session), an SBRS model takes the items of the observed interactions in this session (called the session's context) as input and outputs a list of items as predicted next items. In [5], the researchers take this research even further by introducing a novel framework called COunterfactual COllaborative Session-Based Recommender Systems (COCO SBRS) which takes into account the outer-session causes (OSCs) while making recommendations for session-based systems.

Given the scope of the project and the timeline, I would be implementing the NCF framework in this project and testing for the optimal number of deep layers to build a decently-accurate model for the collaborative filtering phase of the project. I believe using MLP for feature representation is very straightforward and highly efficient, even though it might not be as expressive as auto-encoder, RNN's and CNN's

In [3], the researchers have used Content-based filtering for building the recommendation system. As discussed above, Content-based filtering creates a user profile that reflects his/her preferences. New recommendations are made based on the similarity between the user's preferences and the feature set of the meta-data of the new item. In this paper, the researchers have used Euclidean distance as the similarity measure for the genre matrix. The movies with the closest Euclidean distances are suggested as recommendations

In my project, I plan to use a similar approach but experiment with other measures of similarity like Cosine similarity

4. Design and Implementation of the project

The project consists of two submodules each corresponding to one type of recommendation system. In this section, the model description, challenges, and results of the trained model are discussed. The libraries used, and learning opportunities are also discussed in the later half of the section.

4.1 Collaborative Filtering

A rating dataset is employed for this model since collaborative filtering depends on the user-item interaction matrix. The framework for neural collaborative filtering has been used, as stated in the literature review.

Implicit Feedback

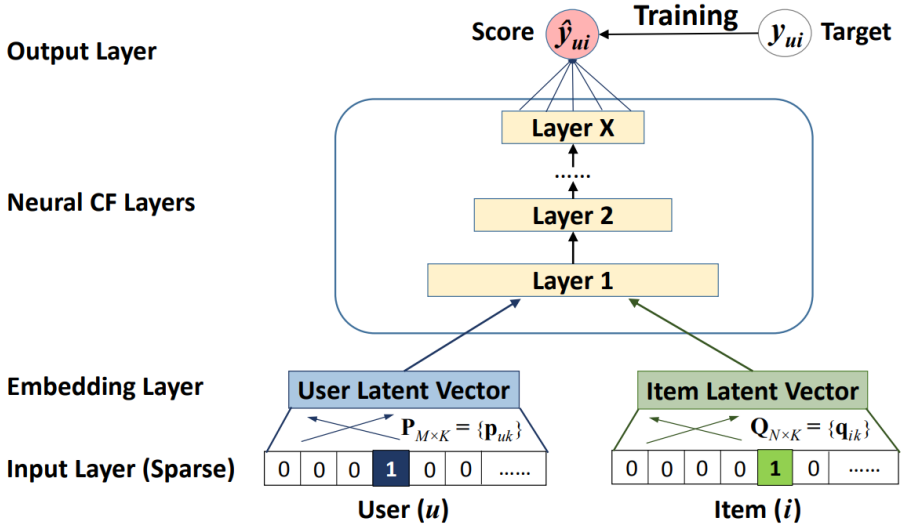
Users' ratings are taken as explicit feedback, so each time a user interacts with an item, he must rate it but in the real world, users don't rate the product often. This is the fundamental downside of the matrix factorization approach, in which we observe that the more sparse the data grows, the less value the matrix factorization method can provide. The NCF framework suggested a novel approach to recommendation systems as a solution to this issue. The researchers suggest using neural networks to forecast this interaction and address the sparsity issue. In NCF, we reduce the rating prediction problem into a more nuanced rating prediction problem of whether the user would interact with the item or not. As a result, the regression issue is reduced to a binary classification issue.

Data Augmentation

We convert the label of the current rating to 1, signifying user interaction, in order to construct a dataset with a binary label to denote user interaction with the object. The issue that arises from this is that there are no negative data points. There are no 0-labeled entries for the neural network to learn from since all ratings have been labeled as 1. As a result, I have employed data augmentation, whereby 4 negative points are generated by randomly selecting movie ID's that the user has not interacted with for every positive interaction. This amounts to 7M data points that will be used to train the model.

Model

Different parameters have been used to test two models. With 7M data points that were produced as previously discussed, the 3-layer model (three hidden layers, one input, and one output) and the 6-layer model were trained. The tower structure has been applied to both models, where the layer closest to the input layer is the widest, and subsequent layers gradually get smaller, creating a tunnel-like structure. This structure's goal is to make the model learn more salient information as it advances through the layers, much as CNNs.



Neural Collaborative Filtering Framework

Embedding Layer:
A layer that learns a low-dimensional representation of high-dimensional input data is known as an embedding layer in a neural network. For Example, In NLP, based on the context in which a word appears in a training corpus, an embedding layer can learn a low-dimensional vector representation of each word in a lexicon.

An embedding layer's goal is to turn high-dimensional, sparse input data, like one-hot encoded vectors, into continuous, dense vectors that may be fed as input to the next layers of a neural network. The learned embeddings can be applied to many tasks, such as language modeling, user modeling, sentiment analysis, and machine translation, to enhance their performance. They can be used to capture semantic associations between words, such as similarity and analogy.

Backpropagation is often used to update the embedding layer during training so that the learned embeddings optimize a certain objective function, like minimizing the prediction error of a downstream task. A hyperparameter that affects how dimensional the learned embeddings are is the size of the embedding layers

As shown in the figure above, the embedding layer is used to convert userId and movieId information into a dense vector which can then be used to pass on to further layers. Here embedding layers of 8 and 16 dimensions were tested

Linear Layer:
In a linear layer, also referred to as a fully connected layer in a neural network, every input neuron is linked to every output neuron by a set of weights and biases. A linear layer converts the input into a new representation by first executing a linear combination of the input values and the weights, then adding a bias term. Once an activation function, such as a sigmoid or ReLU function, has been applied, the layer's output is then returned.

A linear layer's goal is to learn a linear function that converts input data into an output representation that may be used for additional network processing. In a neural network, linear layers are generally utilized in the hidden layers, where they can learn more complicated representations of the input data as the network becomes more complex.

A linear layer can be mathematically defined as $y = Wx + b$, where W is the weight matrix, b is the bias vector, and x is the input vector or matrix. The learnable parameters, weight matrix, and bias vector are tuned during training in order to minimize a loss function.

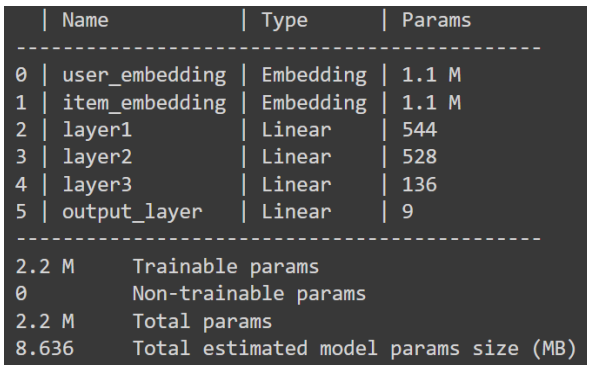
A tower structure has been constructed using linear layers to ensure that later layers learn only the most important features. This relies on the idea of CNNs where Max Pooling layers create the tower structure and ensure that later layers learn the most important features. 3 and 4 hidden linear layers have been tested for performance

Relu activation:
Rectified Linear Unit, or ReLU, is a common activation function in neural networks, particularly in deep learning. It has been demonstrated that with this straightforward and efficient non-linear activation function, neural networks perform better on a variety of tasks, including natural language processing.

In other words, if x is positive, the output of the ReLU function is equal to x, and if x is negative, it is equal to 0. ReLU, then, infuses non-linearity into the network, enabling it to pick up on more intricate correlations between input and output.

Compared to other activation mechanisms, ReLU offers several advantages. First, it just requires straightforward thresholding procedures and it is computationally efficient. Second, it aids in resolving the vanishing gradient issue that deep neural networks sometimes experience. This is because ReLU, unlike other activation functions like tanh and sigmoid, which can become almost flat for big input values, does not saturate in the positive zone.

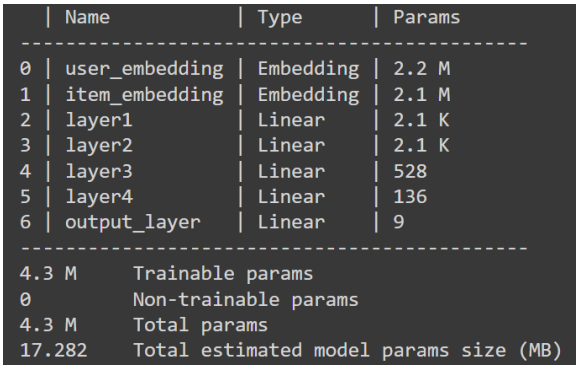
Justification for the usage of Relu [1]: Each neuron must fall inside the boundaries of the sigmoid function, which may limit the model's performance. Moreover, the sigmoid function is known to experience saturation, which causes neurons to cease learning when the output is close to either 0 or 1. Tanh is a more advantageous option and has gained popularity, but because it may be viewed as a rescaled version of the sigmoid (tanh(x/2) = 2(x) 1), it only partially resolves the problems of the sigmoid. ReLU, which has been shown to be non-saturated and is more biologically plausible, favors sparse activations since it works well with sparse data and reduces the likelihood that the model would be overfit.



	Name	Type	Params
0	user_embedding	Embedding	1.1 M
1	item_embedding	Embedding	1.1 M
2	layer1	Linear	544
3	layer2	Linear	528
4	layer3	Linear	136
5	output_layer	Linear	9

2.2 M	Trainable params
0	Non-trainable params
2.2 M	Total params
8.636	Total estimated model params size (MB)

The model with 3 hidden layers



	Name	Type	Params
0	user_embedding	Embedding	2.2 M
1	item_embedding	Embedding	2.1 M
2	layer1	Linear	2.1 K
3	layer2	Linear	2.1 K
4	layer3	Linear	528
5	layer4	Linear	136
6	output_layer	Linear	9

4.3 M	Trainable params
0	Non-trainable params
4.3 M	Total params
17.282	Total estimated model params size (MB)

The model with 6 hidden layers

Training:

Seven Million entries were used to train the models. The outcomes of the two models have been compared using various epochs. The dense vectors of the user and item are concatenated to form a 16 or 32-sized vector which is fed to the neural network to capture relationships between them. The interaction here is largely based on the generation of the dense vector as it decides the orientation of the vector and correlation between items

```python
def forward(self, user_input, item_input):
  dense_user = self.user_embedding(user_input)
  dense_item = self.item_embedding(item_input)
  vector = torch.cat([dense_user, dense_item], dim=-1)

  # Results from various posts and research papers
  # The sigmoid function restricts each neuron to be in (0,1), which may limit the model
  # Even though tanh is a better choice and has been widely adopted it only alleviates t
  # ReLU, which is more plausible and proven to be non-saturated, it encourages sparse a
  vector = nn.ReLU()(self.layer1(vector))
  vector = nn.ReLU()(self.layer2(vector))
  vector = nn.ReLU()(self.layer3(vector))

  # sigmoid is the same as softmax. The better choice for the binary classification is t
  pred = nn.Sigmoid()(self.output_layer(vector))

  return pred
```

PyTorch and PyTorch-lightning modules like Dataset, Dataloader, and Trainer have been used to make the training process easier and simpler.

```python
from torch.utils.data import Dataset, DataLoader

class TrainingData(Dataset):
  def __init__(self, train_ratings):
    # The input has been separated into separate variables so that the later model can use these items to convert to embeddings.
    # Also, each input is sent as a tensor to the successive models
    self.users, self.movies, self.labels = self.get_data(train_ratings)

  def __len__(self):
    # The __len__ function returns the number of samples in our dataset
    return len(self.users)

  def __getitem__(self, idx):
    # Return both inputs (user and item) and the output (target indicating whether the user interacted with the item or not)
    return self.users[idx], self.movies[idx], self.labels[idx]

  def get_data(self, ratings):
    return torch.tensor(ratings['userId']), torch.tensor(ratings['movieId']), torch.tensor(ratings['label'])
```

```
# Creating the model and the trainer
model = CollaborativeFiltering(augmented_dataset, movie_genres, data_loader)
trainer = pl.Trainer(gpus=1, max_epochs=20, enable_progress_bar=True)
```

```
/usr/local/lib/python3.8/dist-packages/pytorch_lightning/trainer/connectors/accelerator_connector.py:478: LightningDeprecationWarning: Setting `Trainer(gpus=1)` is deprecated
  rank_zero_deprecation(
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
```

```
trainer.fit(model)
trainer.save_checkpoint('/content/drive/MyDrive/small_dataset/checkpoint_3layer_regularizer.ckpt')
```

```
INFO:pytorch_lightning.utilities.rank_zero:You are using a CUDA device ('NVIDIA A100-SXM4-40GB') that has Tensor Cores. To properly utilize them, you should set `torch.set_fl
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name           | Type      | Params
-------------------------------------------
0 | user_embedding | Embedding | 1.1 M
1 | item_embedding | Embedding | 1.1 M
2 | layer1         | Linear    | 544
3 | layer2         | Linear    | 528
4 | layer3         | Linear    | 136
5 | output_layer   | Linear    | 9
-------------------------------------------
2.2 M     Trainable params
0         Non-trainable params
2.2 M     Total params
8.636     Total estimated model params size (MB)
/usr/local/lib/python3.8/dist-packages/pytorch_lightning/trainer/connectors/data_connector.py:224: PossibleUserWarning: The dataloader, train_dataloader, does not have many w
  rank_zero_warn(
Epoch 19: 100% 3496/3496 [02:01<00:00, 28.85it/s, loss=0.0092, v_num=16]
```

Metric:

HR@10, also known as Hit Ratio at 10, is a performance indicator frequently used to assess the precision of a recommendation system. The percentage of test situations in which the algorithm accurately predicts at least one pertinent item among the top 10 recommendations is measured by HR@10.

The list of test cases and the list of suggested items for each test case must first be defined in order to calculate HR@10. For example, consider the scenario where we have 100 test cases and 10 items are suggested for each test case.

Next, We count the number of test cases that contain at least one of the top 10 suggested items. You can accomplish this by contrasting the sets of suggested items with the sets of pertinent items for each test scenario. The test case is deemed successful if at least one of the suggested items is present in the collection of pertinent items.

Finally, dividing the number of hits by the total number of test cases we get HR@10.. Since we need percentage we multiply by 100

$$HR@10 = (number\ of\ hits/total\ number\ of\ test\ cases) * 100$$

Diversity Regularizer:
Diversity in a recommendation system describes how many different items, as opposed to only a small number of items, are recommended to consumers. A system that offers a variety of recommendations will put out a variety of items that are distinct from one another and may be appealing to consumers with various preferences and tastes

The relevance of diversity in a recommendation system is twofold. By providing a greater variety of options, it can first increase user happiness and engagement with the system, decreasing the likelihood of boredom or tiredness. Second, it can aid in addressing the popularity bias issue, which occurs when the system primarily suggests well-known, popular items to users while ignoring the "long tail" of less-popular items that might be of interest to some users.

The method for making sure diversified content is being recommended is to make recommendations based on the tastes of users who share similar interests while also making sure the recommended items are sufficiently distinct to avoid homogeneity.

```
def diversity_loss(self, y_true, y_pred, movie_ids):
  # Adding the diversity loss as a regularizer to the log loss function
  # This has been added to enhance diversity of the model predictions
  alpha = 10**-3
  movie_ids_list = movie_ids.tolist()
  positives = movie_ids_list[np.argsort(y_pred.tolist())[::-1][:10]]
  batch_grid = self.genre_grid.loc[positives]
  similarity = batch_grid.corr()
  diversity_regularizer = (similarity.sum()).sum()

  # Alpha here is multiplied to soften impact of the size of loss
  return alpha * diversity_regularizer
```

Testing and Results:
Comparing the testing procedure for the recommendation system to the general NN issues reveals significant differences. We employ the leave-one-out method. The most recent interaction is kept on file for each user. The important thing to note is that we don't require the user to interact with each and

every suggestion in the list. As long as the user interacts with at least one item on the list, we consider the experiment a success.

To put this in the context of the project's objectives, the user's most recent interaction is selected as the test data, while the remaining interactions are used for training. If the test item appears on the list when we use the model to get the top 10 recommendations, we classify that calculation as a success.

```
from functools import partial
tqdm = partial(tqdm, position=0, leave=True)

test_user_item_set = set(zip(test_ratings['userId'], test_ratings['movieId']))
test_dataset = pd.read_csv('/content/drive/MyDrive/small_dataset/augmented_test_dataset.csv')

hits = []
user_ids = test_dataset['userId'].unique()
for user_id in tqdm(user_ids):
    test_item = test_ratings[test_ratings['userId']==user_id]['movieId'].iloc[0]
    user_df = test_dataset[test_dataset['userId'] == user_id].reset_index()
    data_loader = DataLoader(TestingData(user_df), batch_size=100, num_workers=4, shuffle=False)

    # Returns a list of dictionaries, one for each provided dataloader containing their respective predictions
    predictions = model(torch.tensor(user_df['userId']), torch.tensor(user_df['movieId']))
    # To convert to numpy array and solve issue: Can't call numpy() on Tensor that requires grad. Use tensor.detach().numpy() instead
    predictions = predictions.detach().numpy()

    # To solve : Buffer has wrong number of dimensions (expected 1, got 2) because dimensions of predictions are (100, 1)
    # Reference: https://deeplizard.com/learn/video/fCVuiW9AFzY
    predictions = np.squeeze(predictions)

    # Since we need the movieId,
    top_10 = set(user_df.iloc[np.argsort(predictions)[::-1][:10]]['movieId'])

    hits.append(1) if test_item in top_10 else hits.append(0)

print(f'Hit Ratio @ 10 is {np.average(hits)}')
```

```
100%|██████████| 14315/14315 [01:09<00:00, 206.72it/s]Hit Ratio @ 10 is 0.5249039469088369
```

The results show that the 3-layer model has achieved a decent HitRatio@10 of 0.524 while the 4-layer model has achieved 0.593. The base paper claims to have achieved 0.67 on the MovieLens 1M dataset. The current model seems to have achieved close results.

```
hits = []
user_ids = test_dataset['userId'].unique()
for user_id in tqdm(user_ids):
    test_item = test_ratings[test_ratings['userId']==user_id]['movieId'].iloc[0]
    user_df = test_dataset[test_dataset['userId'] == user_id].reset_index()
    data_loader = DataLoader(TestingData(user_df), batch_size=100, num_workers=4, shuffle=False)

    # Returns a list of dictionaries, one for each provided dataloader containing their respective predictions
    predictions = model(torch.tensor(user_df['userId']), torch.tensor(user_df['movieId']))
    # To convert to numpy array and solve issue: Can't call numpy() on Tensor that requires grad. Use tensor.detach().numpy() instead
    predictions = predictions.detach().numpy()

    # To solve : Buffer has wrong number of dimensions (expected 1, got 2) because dimensions of predictions are (100, 1)
    # Reference: https://deeplizard.com/learn/video/fCVuiW9AFzY
    predictions = np.squeeze(predictions)

    # Since we need the movieId,
    top_10 = set(user_df.iloc[np.argsort(predictions)[::-1][:10]]['movieId'])

    hits.append(1) if test_item in top_10 else hits.append(0)

print(f'Hit Ratio @ 10 is {np.average(hits)}')
```

```
100%|██████████| 14315/14315 [01:07<00:00, 211.16it/s]Hit Ratio @ 10 is 0.5700314355571079
```

With regularization applied, the HitRatio@10 of the model has increased to 0.57 for the 3-layer model. This shows that adding the diversity regularizer has helped but there is still the need for a better way to calculate the diversity loss. I would leave that as the future work of the project which can help suggest diversified content to users.

The number of layers	HitRatio@10
3-layer model(without regularization)	0.524
3-layer model(with regularization)	0.57
4-layer model(without regularization)	0.59

Challenges

- **Data Augmentation**

  Due to huge amout of data, I faced issues with loading the data and processing it initially. It was due to non-optimized code that I used initally to augment the data. I had created unnecessary sets and lists which would smong the process, making it super slow.

```
from concurrent.futures import ThreadPoolExecutor

def augment(users, data, movies):
    subset = pd.DataFrame(columns=data.columns)
    for i, user_id in enumerate(users):
        movies_interacted_with = set(data[data['userId'] == user_id]['movieId'].values)
        movies_not_interacted_with = movies - movies_interacted_with

        for _ in range(20):
            unsuccessful_recommendation = np.random.choice(tuple(movies_not_interacted_with))
            movies_not_interacted_with.remove(unsuccessful_recommendation)
            subset = subset.append({'userId': user_id, 'movieId': unsuccessful_recommendation, 'target': 0}, ignore_index=True)

    subset = subset.sample(frac=1).reset_index(drop=True)
    return subset
```

```
from concurrent.futures import ThreadPoolExecutor

data = train_ratings[['userId', 'movieId']]
data['target'] = 1

unique_users = data['userId'].unique()
users = len(unique_users)
unique_movies = set(data['movieId'].unique())

futures = []
with ThreadPoolExecutor(max_workers=10000) as executor:
    for i in range(users//100 + 1):
        block_start = i * 100
        block_end = min(users, 100 * (i+1))
        print('Submitting ', i)
        futures.append(executor.submit(augment, unique_users[block_start: block_end], data, unique_movies))
```

```
for i, future in enumerate(futures):
    data = pd.concat([data, future.result()], ignore_index=True)
    print(f'Created negative data for block {i}. {len(futures) - i} left')

data.to_csv('augmented_dataset.csv', index=False)
```

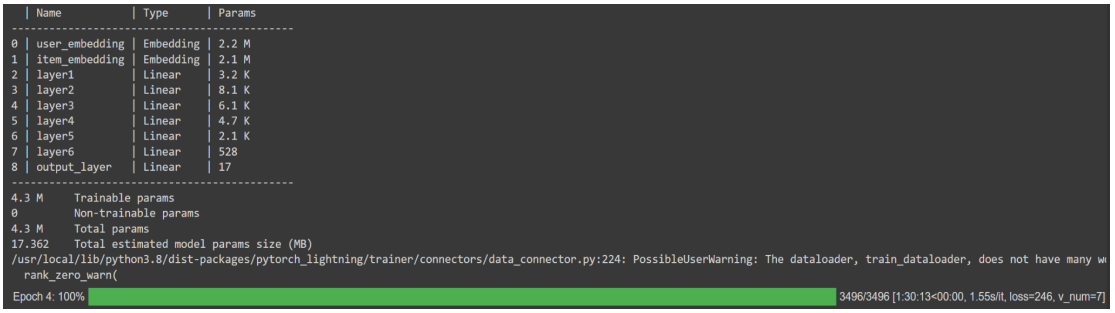
I later realized that what Colab does best is chunking of data and process switching to manage memory. I then used a simple loop which run smoothly without stalling my chrome.

Also, later I placed all the data augmentation work in a separate file and created the augmented datasets as csv so that data augmentation does not happen each time I try to train the model due to its time consuming nature

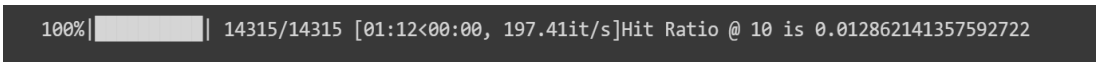
- **Colab session timeout**
  As the data was huge, it took hours to train just 5 epochs. During this process, Colab always disconnected the session, never letting me complete the training process for the complete data. I reduced the data from 20M to 7M and also subscribed to Colab premium to solve the problem of timeouts

- **Deciding the Model Architecture**
  Model architecture is very crucial for the model to be able to capture important features from data and propagate the results to further layers. While deep models have the upper hand when it comes to accuracy of predictions, they definitely require more training epochs and need more time to tune the millions of the weights (parameters).
  After careful study of different papers, I was able to drill down the options are tried with 3 and 4 layers. Future work could include having more layers to achieve better results

- **Designing the diversity regularizer**
  Diversity regularizer was the new idea I had that I wanted to try on the model. I tried different methods like dot products, transformations etc. all of which resulted in unpredictable results and some resulted in HitRatio@10 being 0.0



Un-optimized diversity regularizer function resulted in each epoch taking 1.5 hours



Later, I wanted to try clustering to calculate similarity and during that process of implementation, I realized correlation could be a good metric. To overcome the problem of huge regularization terms, I included alpha=0.001.

4.2 Content-Based Recommendation System

Content-based recommendation systems rely on the meta-data of the objects or items and locate thematically linked objects. Thus, the model completely relies on the similarity measure being used.

Similarity Measure

A similarity measure between two non-zero vectors in an n-dimensional space is called cosine similarity. It is frequently used to compare the direction and magnitude of two vectors.

The range of cosine similarity, from -1 to 1, determines the cosine of the angle formed by the two vectors. The two vectors are said to be identical if their values are 1, and to have opposing directions if they have values of -1. A value of 0 represents the vectors as being orthogonal. (That is, there is no similarity between them).

Model

In general, the genres are represented as 1 or 0 for indicating whether the genre is relevant to the movie or not. In this project, I have used the average rating of the project as the relevance factor to indicate the relevance of the genre. This can help create more accurate and closely related objects by awarding the high-rated movies

```
# Expand the genres into columns
genres = set()
for index, row in content.iterrows():
  for value in row.genres.split('|'):
    genres.add(value)

# In general, the values of the columns are set to 0 or 1 based on the genre list available
# But in this case, I have used the rating as the value so that a stronger linkage can be found
content[[list(genres)]] = 0
for index, row in content.iterrows():
  for column in genres:
    content.loc[index, column] = row.rating if column in row.genres else 0

content.head()
```

	title	genres	rating	Animation	Thriller	Sci-Fi	Comedy	Mystery	Fantasy	War	...	Adventure	IMAX	Documentary	Children
1	Toy Story	Adventure|Animation|Children|Comedy|Fantasy	3.934426	3.934426	0.0	0.0	3.934426	0.0	3.934426	0.0	...	3.934426	0.0	0.0	3.934426
2	Jumanji	Adventure|Children|Fantasy	3.267199	0.000000	0.0	0.0	0.000000	0.0	3.267199	0.0	...	3.267199	0.0	0.0	3.267199
3	Grumpier Old Men	Comedy|Romance	3.005319	0.000000	0.0	0.0	3.005319	0.0	0.000000	0.0	...	0.000000	0.0	0.0	0.000000
4	Waiting to Exhale	Comedy|Drama|Romance	2.571429	0.000000	0.0	0.0	2.571429	0.0	0.000000	0.0	...	0.000000	0.0	0.0	0.000000

As for the similarity measure, I have used the dot product of the genre matrix created above to rank the objects. The top 10 recommendation results are listed below for the movie ‘Toy Story’

```
# I have built Content-based filtering as a purely mathematical model using similarity of its genres
def get_recommendations(movie_title, n=20):
  movie_id = content[content['title'] == movie_title].index[0]
  movie_of_interest = movie_genres.loc[movie_id]

  result = movie_genres.dot(movie_of_interest)

  recommendations_index = result.sort_values(ascending=False)[:n].index
  recommendations = content.loc[recommendations_index]
  return recommendations
```

```
# Getting results/recommendations for `Toy Story`
result = get_recommendations('Toy Story')
print(result['title'])
```

```
80158                    Cartoon All-Stars to the Rescue
131248                                  Brother Bear 2
78499                                     Toy Story 3
1                                           Toy Story
26340                   Twelve Tasks of Asterix, The
4886                                    Monsters, Inc.
3114                                      Toy Story 2
108932                                 The Lego Movie
4306                                            Shrek
4016                      Emperor's New Groove, The
```

Packages Used:

Numpy, Pandas, DataLoader from torch, torch.nn, pytorch, pytorch_lightning, tqdm

Learnings:

- **Understanding of Recommendation Systems: The secret sauce**
  Most part of my project has been spent on research and trying to understand how different models are implemented and used. I have developed a good understanding of different generations of recommenders starting from knowledge-based and rule-based, to statistical models, to the current generation of deep learning models. Through this project, I have been able to explore and learn about Nobel-prize award-winning theory Nudge theory, and Personality Psychology.

- **Know-how of various advancements**
  In the pursuit of learning about recommendation systems, I have explored new models proposed in recent research. Session-based recommendation systems, Autoencoder-based systems, are a few I have come across

- **Deep learning fundamentals and Packages**
  - Data Augmentation
  - Model development
  - Pytorch and Pytorch-lightning

- **Trade-offs between activation functions and optimizers**
  Activation functions are a key component of artificial neural networks and are used to introduce non-linearity in the output of a neuron. Different activation functions have different properties, and the choice of activation function can have a significant impact on the performance of a neural network. Here are some trade-offs between different activation functions:
  - Sigmoid: The sigmoid activation function produces an output that is always between 0 and 1, making it useful for binary classification problems. However, the gradient of the sigmoid function is small for large inputs, which can lead to vanishing gradients and slow convergence.
  - Tanh: The hyperbolic tangent (tanh) activation function produces an output that is always between -1 and 1, making it useful for classification problems with symmetric output. However, like sigmoid, it also suffers from the problem of vanishing gradients.
  - ReLU: The rectified linear unit (ReLU) activation function is widely used in deep learning because it is computationally efficient and avoids the vanishing gradient problem. However, ReLU can suffer from the problem of "dying ReLU" where a large fraction of the neurons become inactive and produce zero output. This can be mitigated by using variants of ReLU such as leaky ReLU or ELU.
  - Softmax: The softmax activation function is commonly used in the output layer of a neural network for multi-class classification problems. It converts the output of each neuron into a probability distribution over the classes. However, it requires all the classes to be mutually exclusive and suffers from the problem of "softmax saturation" when the outputs of the neurons are very large or very small.

- **Troubleshooting**
- **Experimentation**

5. Future Work

Deep Learning Models for Content-Based Filtering:

For now, I have applied mathematical models for Content-Based Filtering but the recommendation system can be much improved if we apply deep learning models to Content based Filtering.

Implement NeuMF model:

To capture both the non-linear and linear relationships between users and items, NeuMF integrates two different types of neural networks, namely Matrix Factorization (MF) and Multi-Layer Perceptron (MLP). By joining the high-dimensional feature representations learnt by MLPs with the low-dimensional latent components generated through matrix factorization, NeuMF integrates these two methods. NeuMF is able to produce more accurate suggestions as a result of being able to capture both linear and non-linear associations between users and objects.

Session based recommendations

In addition to Collabarative and Content Based Filtering, session based recommendation would help improve the recommendation systems. Session based recommendation recommends the user based on the items or the sequence of items they have interacted during their current session

References:

[1]. He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T. (2017). Neural Collaborative Filtering. *ArXiv*. https://doi.org/10.48550/arXiv.1708.05031

[2]. Liang, D., Krishnan, R. G., Hoffman, M. D., & Jebara, T. (2018). Variational Autoencoders for Collaborative Filtering. ArXiv. https://doi.org/10.48550/arXiv.1802.05814

[3]. Reddy, Srs & Nalluri, Sravani & Kunisetti, Subramanyam & Ashok, S & Venkatesh, B. (2018). Content-Based Movie Recommendation System Using Genre Correlation.

[4]. Recommendation System Series Part 2: The 10 Categories of Recommendation Systems, https://towardsdatascience.com/recommendation-system-series-part-2-the-10-categories-of-deep-recommendation-systems-that-189d60287b58

[5]. Song, W., Wang, S., Wang, Y., Liu, K., Liu, X., & Yin, M. (2023). A Counterfactual Collaborative Session-based Recommender System. ArXiv. https://doi.org/10.48550/arXiv.2301.13364

[6] Recommendation Systems Introduction
https://medium.com/towards-data-science/deep-learning-based-recommender-systems-3d120201db7e

[7] 7 Types of Hybrid Recommendation System, Jeffery Chiang
https://medium.com/analytics-vidhya/7-types-of-hybrid-recommendation-system-3e4f78266ad8

[8] Visualization of Movie Lens Dataset
https://towardsdatascience.com/comprehensive-data-explorations-with-matplotlib-a388be12a355

[9] Deep Learning Neural Network implementation of Collaborative Filtering
https://towardsdatascience.com/introduction-to-recommender-systems-2-deep-neural-network-based-recommendation-systems-4e4484e64746

[10] PyTorch docs and documentation: https://pytorch.org/docs/stable/index.html