Recommender System

Project-5 Extension: Recommender System for Starbucks Using XGBoost

Monalika Pradhan
College of Engineering
Northeastern University
Boston, MA
pradhan.mon@northeastern.edu

Abstract:

Globally recognized, Starbucks stands as a prominent coffeehouse chain from America, based in Seattle, Washington, epitomizing the second wave of American coffee culture. With more than 30,000 locations around the world, the company prioritizes superior customer service and experience. To boost customer interaction, Starbucks provides a complimentary app that enables online orders, special offers, and the collection of rewards. The goal of the company is to customize the app's experience by forecasting and adapting offers to align with each customer's preferences. Starbucks presents three kinds of promotions: Buy One Get One (BOGO), discounts, and informational offers. These offers are distributed through different mediums, and customers have the option to disregard, view, or respond to them. The aim is to examine past data that includes user activities, profiles, purchases, and rewards earned from responding to these offers. Analyzing this data will assist in crafting a model designed for personalized offer recommendations, which could lead to a higher engagement with these offers. This initiative involves the development of a main recommendation engine, an alternate model geared towards enhanced performance, and a comparative analysis of their efficiencies. This is a an implementation of a paper linked here: https://paperswithcode.com/paper/regularizing-matrix-factorization-with-user

Datasets:

portfolio.json: This file holds offer IDs along with details about each offer like its duration, type, etc. The attributes in this file include:

Sr.no	Attributes	Description	Data Type

1.	id	The primary key/ id that uniquely identifies a particular offer	String
2.	offer_type	The type of offer (e.g., BOGO, discount, informational).	String
3.	difficulty	The minimum spend required to complete an offer.	Int
4.	reward	The reward for completing an offer.	Int
5.	duration	The number of days an offer is available.	Int
6.	Channels	Various channels through which the offer is communicated.	list of strings

profile.json: This contains demographic information of each customer. The variables include:

Sr.no	Attributes	Description	Data Type
1.	age	Customer's age.	Int
2.	became_member_on	The date when the customer registered an app account	Int
3.	gender	Customer's gender (M, F, or 'O' for other).	String
4.	id	Customer ID.	String
5.	income	Customer's income.	Float

transcript.json: This file records transaction details, including offers received, viewed, and completed. The variables are:

Sr.no	Attributes	Description	Data Type
1.	event	A description of the record (e.g., transaction, offer received/viewed).	Int
2.	person	Customer ID.	String
3.	time	Time in hours from the start of the test, starting at t=0.	Int

4.	value	Contains either an offer ID or a transaction amount, depending on the record	String

Creating a User Behavior Recommendation Matrix

To analyze customer behavior, it's crucial to identify the customer responses to specific offers. Our approach involves constructing a recommendation matrix for customers by integrating all the available datasets. This process will use pre-processed data according to the outlined method.

Initially, we'll pull the 'value' field from the transcript dataset and link it with the portfolio dataset. The portfolio dataset is rich with essential details about each unique offer. Subsequently, we merge this combined data with the information in the profile dataset, matching on the 'person' ID field. This comprehensive merging will reveal the extent of offers received by each customer, including the specific channels used for each offer.

Moreover, this merged data will encompass information about the cost in rewards for each offer, its duration, and type. Finally, we plan to refine the dataset by focusing on the most recent customer interactions, with the filtering specifics outlined in the Feature Engineering section.

Feature Engineering:

In order to effectively train our machine learning model, we need to transform both categorical and continuous data into a format that is comprehensible to the model. Key steps include:

1. Encoding the 'channel' field into binary values. For instance, in the case of a mobile channel, '0' will indicate its non-use in an offer's promotion, while '1' will signify its use.

2. Extracting the duration of each customer's membership, represented as 'member_days'. This metric is vital for accurate predictions.

3. Implementing binning techniques on the 'income' field to enhance model performance.

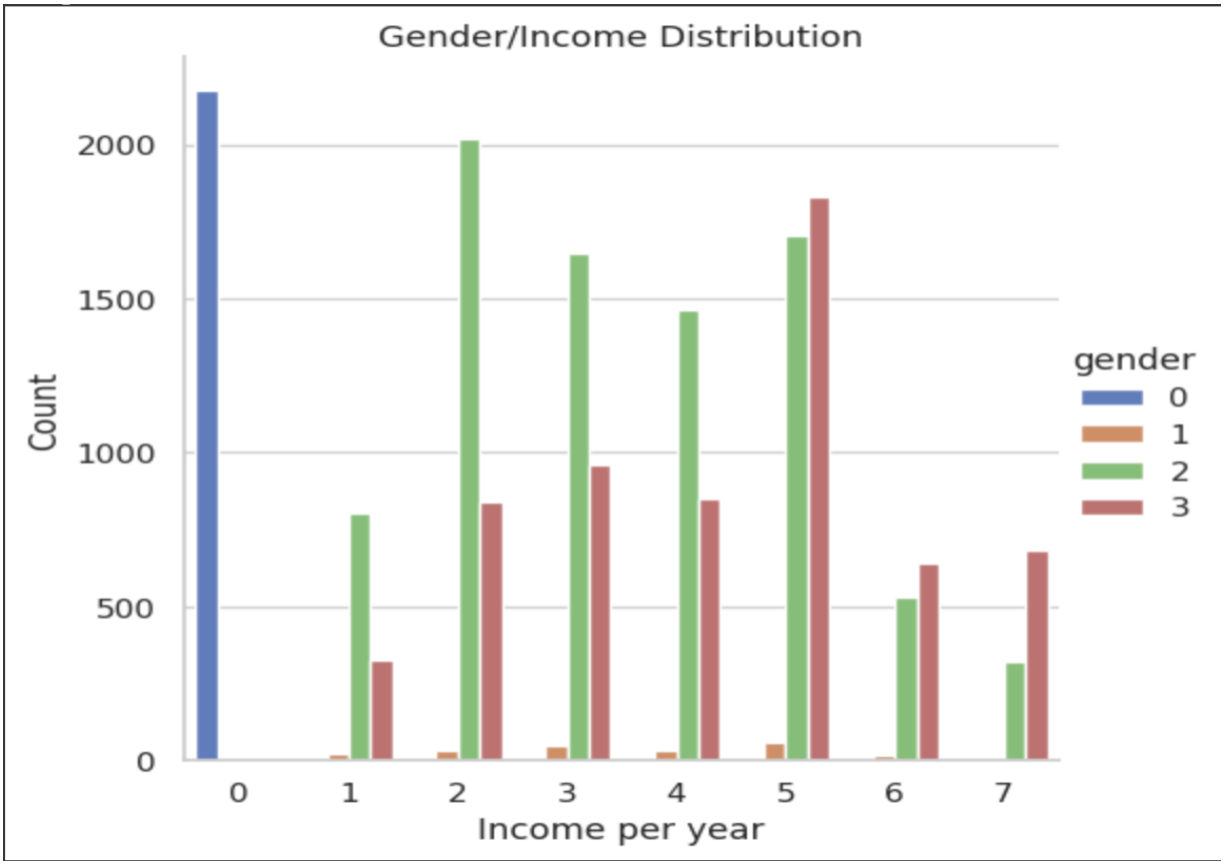
4. Mapping gender data into categorical values. The mapping will be as follows: 'X' for unspecified gender (often associated with unreported income), 'M' for male, and 'F' for female, coded as 0, 1, 2, and 3, respectively. Unspecified gender data, usually marked as missing, will be treated as a distinct category to glean insights, especially since some customers choose not to disclose their gender or income. These customers often engage with offers, indicating potential to convert them into active customers.

5. Processing the 'event' field to understand customer reactions to offers. This field includes 'offer received' for offers received by customers, 'offer viewed' for at least viewed offers, and 'offer completed' for offers that led to a purchase.

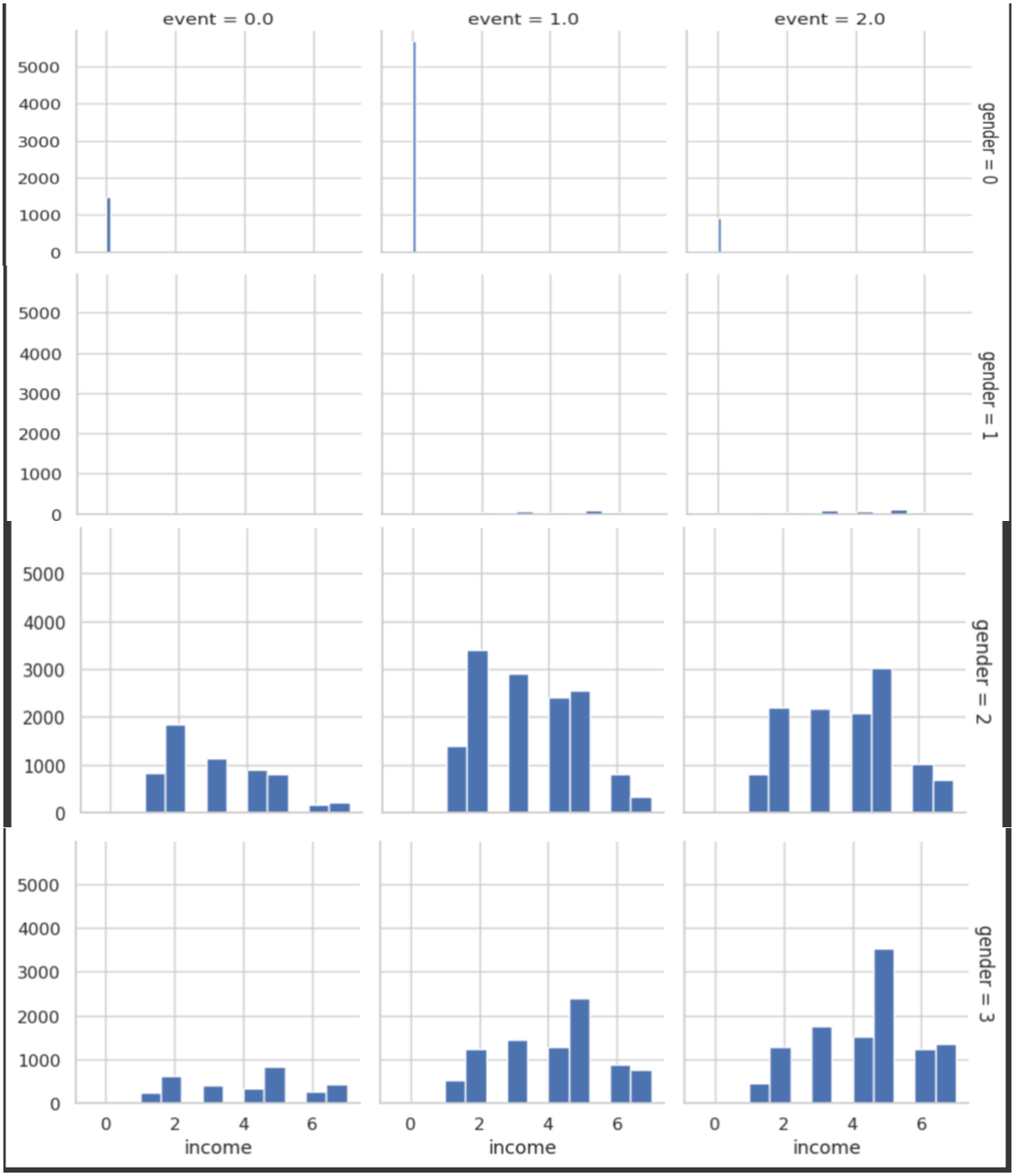
The final step involves extracting the most recent action taken by a customer in response to each offer, whether it was ignored, viewed, or responded to. Keeping only the latest customer action is crucial for building an accurate customer recommendation matrix.

Exploratory Data Analysis (EDA)

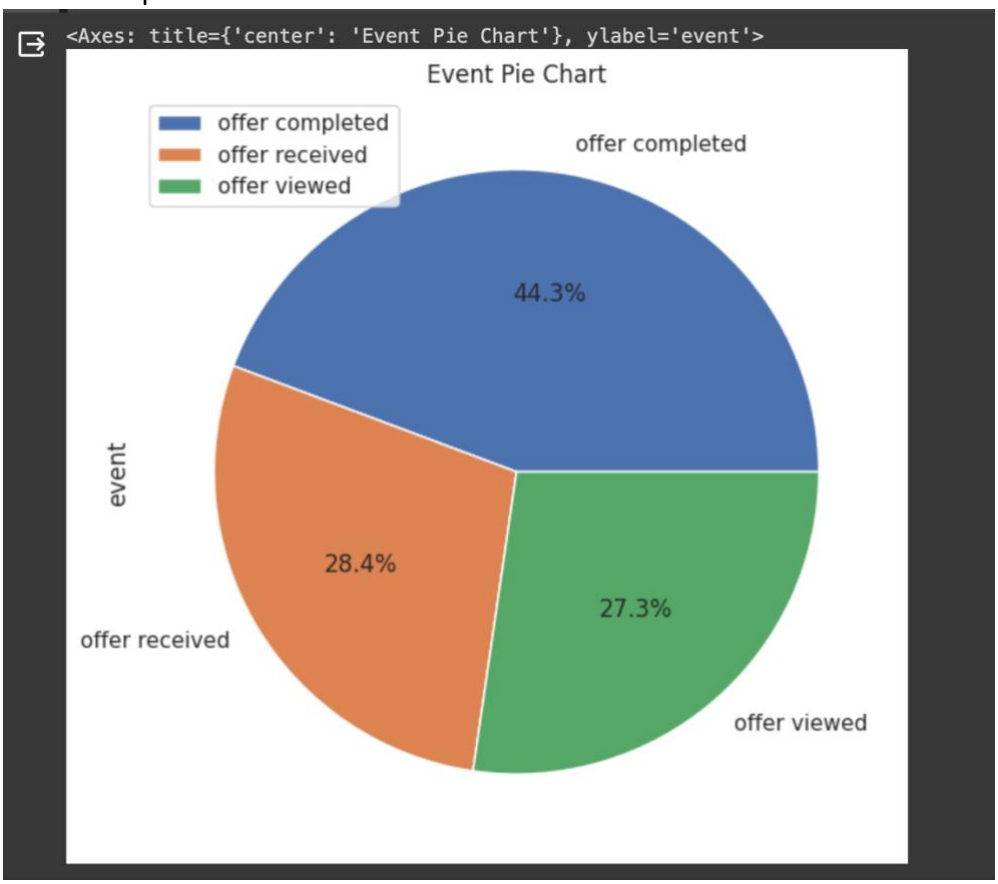
The below graphs shows us how the data is categorized based on 'income' and 'gender', with the objective of counting the number of occurrences within each distinct category. Utilizing Seaborn, a popular data visualization library in Python, a bar chart is generated. This chart effectively illustrates the distribution of counts across various income levels, with an additional layer of differentiation based on gender.



Creating more detailed graphs to examine how each gender category, including those who chose to remain anonymous, reacts to various offers.

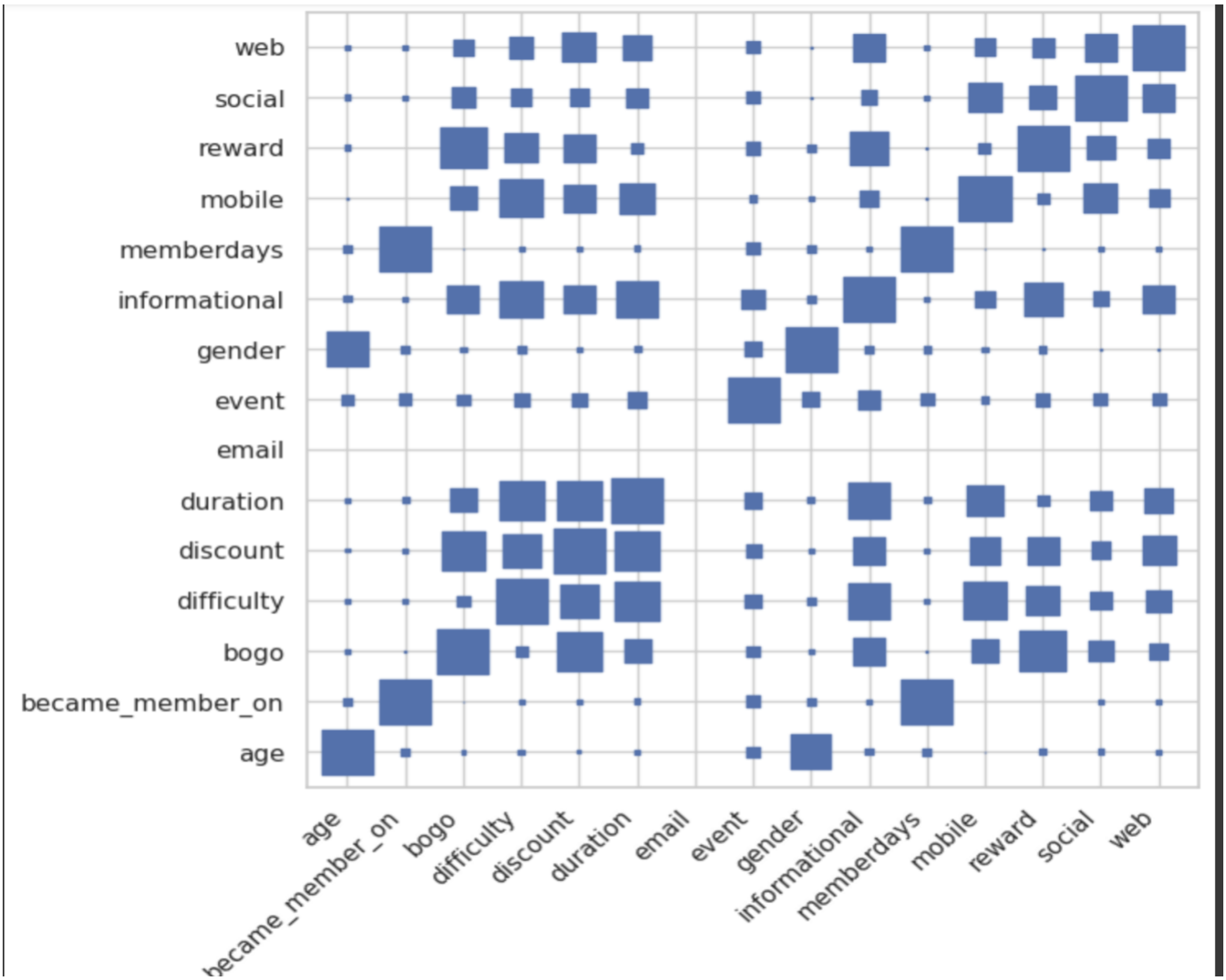


In the analysis, a pie chart is generated to visualize the distribution of the first 1000 user event actions within the 'user_actions' DataFrame. These events are categorized into three types: 'offer received', 'offer viewed', and 'offer completed', corresponding to numerical values 0, 1, and 2. The chart, with a dimension of 7x7 inches, effectively illustrates the proportional representation of each event type. It includes features like a descriptive title "Event Pie Chart", percentages for each event type calculated automatically, and a legend for ease of interpretation. This visualization aids in understanding the relative frequency of different types of user responses to offers.

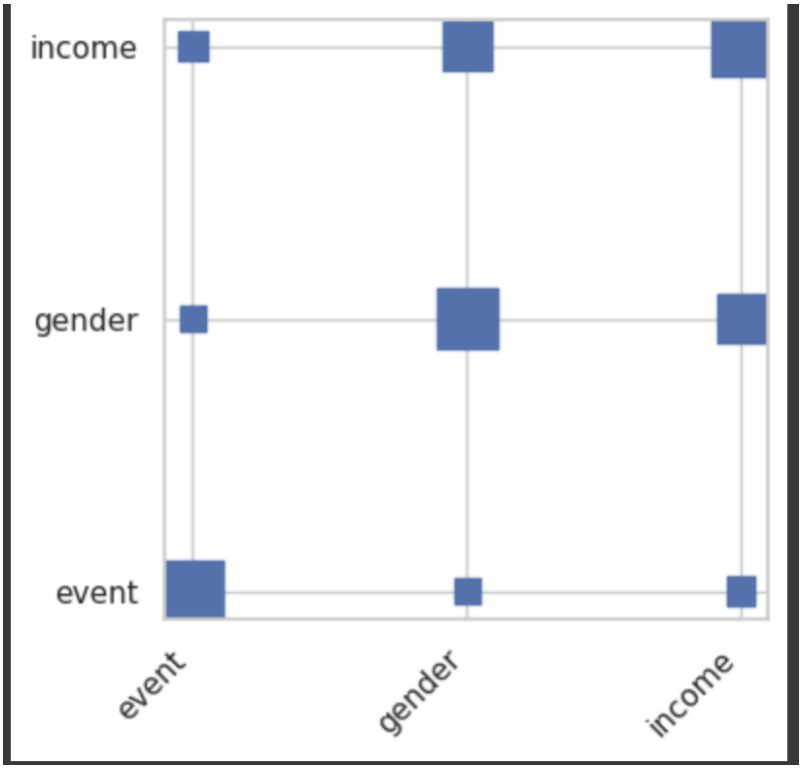


Correlations:

Used heatmap function in Python for creating heatmaps from data, with customization options like figure size and file name. This function maps the unique values of x and y columns to integer coordinates and sizes the points in the scatter plot based on a given size parameter. After plotting, it labels the axes with sorted column names. The function is then used to generate a heatmap visualizing the correlations between various offer and user specifications in a dataset, saving the image with a specified file name.


web
social
reward
mobile
memberdays
informational
gender
event
email
duration
discount
difficulty
bogo
became_member_on
age
age
became_member_on
bogo
difficulty
discount
duration
email
event
gender
informational
memberdays
mobile
reward
social
web

This is the correlation between 'income', 'gender', and 'event' columns in a DataFrame, then reshapes it for a heatmap visualization. It generates a heatmap, saved under a specified file name, with point sizes representing the absolute values of the correlations.



Algorithms Used:

In our study, we delved into the realm of recommendation systems, specifically focusing on the product-based collaborative filtering approach. This method hinges on analyzing the similarities in user responses to various offers, leveraging existing data. Key to this approach is the use of embedding matrices, which act as lookup tables for generating feature vectors representing both users and offers. These vectors, treated as features, are then integrated into a sophisticated multi-layer Neural Network (NN) with Embedding layers. This NN is tailored to encapsulate the user-offer interaction matrix, along with additional parameters that enhance the model's capability to derive insights from the data.

Our experiments extended to various configurations of batch sizes, dropout rates, and the exploration of different optimizers and learning rates. Notably, the Stochastic Gradient Descent (SGD) optimizer, augmented with momentum, demonstrated superior performance, achieving the lowest loss in both training and validation phases. This was exemplified in a recommendation engine that we tested, where a model with the same topology and batch size attained an accuracy of 68.3% using SGD, in contrast to only 38.3% when employing the Adam optimizer, which is commonly preferred in many scenarios. This significant discrepancy underscores the effectiveness of the chosen approach in our recommendation system model.

Implementation of Baseline Model:

The Model class, represents a sophisticated neural network architecture developed using PyTorch, a prominent deep learning framework. This class is structured to accommodate recommendation systems, leveraging the power of embeddings and fully connected layers. Key aspects of the model include:

1. Inheritance and Initialization: Model is derived from PyTorch's nn.Module, ensuring integration with PyTorch's functionalities. It initializes with parameters for the number of users (n_users), items (n_items), the dimensionality of embeddings (embed_dim), output layer dimension (output_dim), a list specifying the sizes of hidden layers (layers), and the dropout probability (p).

2. Embedding Layers: The model features separate embedding layers for users (u_emb) and items (m_emb), each with a specified embedding dimension, facilitating the capture of latent features within these entities.

3. Layer Construction: The neural network is constructed dynamically based on the layers parameter. It consists of a sequence of Linear layers, interspersed with ReLU activation functions, BatchNorm1d for normalization, and Dropout layers for regularization. The network's input size is twice the embedding dimension, accounting for the concatenated user and item embeddings.

4. Weight Initialization: The weights of the first and last linear layers undergo Xavier uniform initialization, while their biases are set to zero. This approach aids in stabilizing the learning process during training.

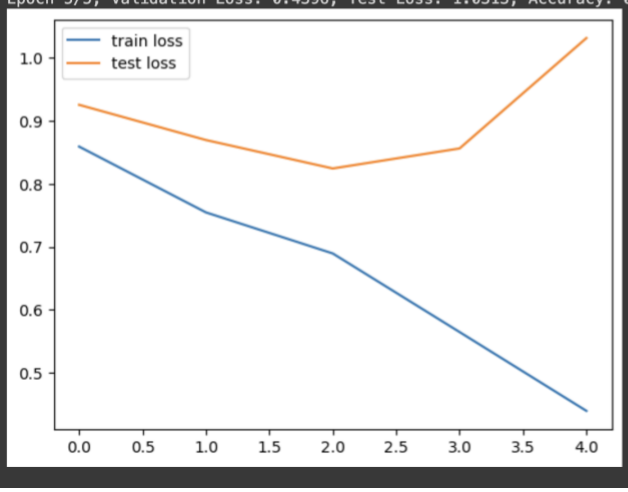
5. Forward Pass Methodology: The forward method delineates the data flow through the model. Both user and item inputs are first transformed via their respective embedding layers, then concatenated. This combined representation is fed through the sequential layers to produce the final output.

Overall, this Model class encapsulates a versatile and robust framework for a neural network-based recommendation system, emphasizing the synergistic use of embeddings and a series of neural network layers to effectively model and predict user-item interactions.

Plotting train and test loss per iteration::

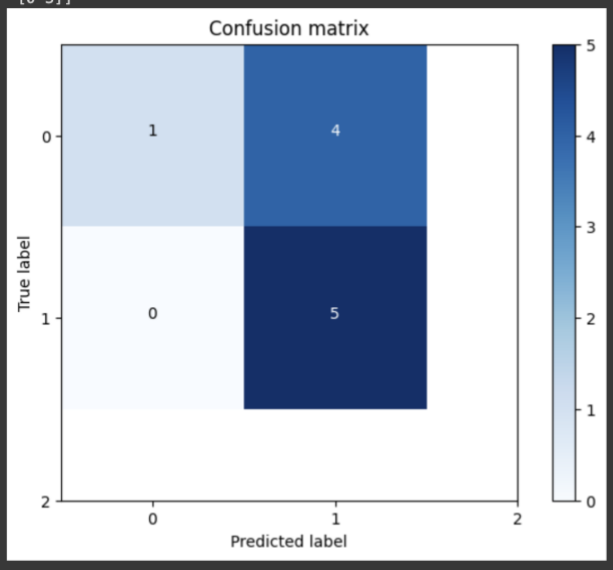
```
train_losses, val_losses = batch_gd(model, criterion, optimizer, train_loader, validation_loader, 5)

# Plot the train loss and test loss per iteration
plt.plot(train_losses, label='train loss')
plt.plot(val_losses, label='test loss')
plt.legend()
plt.show()
```

```
Epoch 1/5, Validation Loss: 0.8590, Test Loss: 0.9254, Accuracy: 0.8181818181818182, Duration: 0:01:08.434195
Epoch 2/5, Validation Loss: 0.7546, Test Loss: 0.8695, Accuracy: 0.7727272727272727, Duration: 0:01:12.321690
Epoch 3/5, Validation Loss: 0.6894, Test Loss: 0.8243, Accuracy: 0.8181818181818182, Duration: 0:01:13.342689
Epoch 4/5, Validation Loss: 0.5647, Test Loss: 0.8561, Accuracy: 0.8181818181818182, Duration: 0:01:16.585142
Epoch 5/5, Validation Loss: 0.4396, Test Loss: 1.0313, Accuracy: 0.8636363636363636, Duration: 0:01:12.172040
```



Confusion Matrix and Accuracy for baseline Model:

```
Confusion matrix, without normalization
[[1 4]
 [0 5]]
```




```
[22] print("Accuracy so far: " + str(100*accuracy_score(test_y, predicted))+ "%" )
```

```
Accuracy so far: 60.0%
```

Creating a neural network with additional continuous parameters:

In the advanced neural network architecture, Model, I am using PyTorch for recommendation system purposes. This class extends PyTorch's nn.Module and is notable for its capability to handle additional continuous parameters related to users and offers. Key features of the Model class include:

1. Initialization with Extended Parameters: The model is initialized with dimensions for users (n_users), items (n_items), continuous user parameters (n_cont_user), continuous offer parameters (n_cont_offer), embedding dimension (embed_dim), output dimension (output_dim), a list of layer sizes (layers), and dropout probability (p).

2. Embedding and Batch Normalization Layers: It includes embedding layers for users (u_emb) and items (m_emb), as well as batch normalization layers for the continuous parameters of users (bn_cont_u) and offers (bn_cont_o).

3. Dynamic Layer Construction: The model dynamically constructs its layers to accommodate the combined dimensions of user/item embeddings and continuous parameters. This includes a series of Linear layers, BatchNorm1d, Dropout, and ReLU activations, ending with a Linear layer mapping to output_dim.

4. Weight Initialization: Xavier uniform initialization is applied to the weights of the first and last linear layers, and biases are initialized to zero.

5. Forward Method: In the forward pass, user and item embeddings are concatenated with batch-normalized continuous user and offer details. This combined data is then processed through the sequential layers to produce the output.

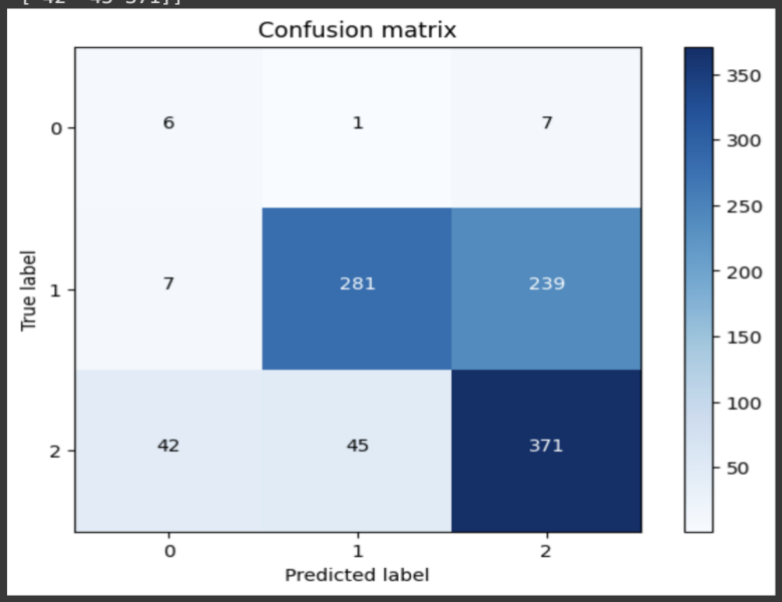
Overall, this Model class represents a comprehensive approach in neural network design for recommendation systems, incorporating both categorical (user/item) and continuous data, offering a more nuanced and potentially accurate prediction capability.

The Architecture of the model is as follows:

```
Model(
  (bn_cont_u): BatchNorm1d(5, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn_cont_o): BatchNorm1d(10, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (u_emb): Embedding(16994, 20)
  (m_emb): Embedding(10, 20)
  (layers): Sequential(
    (0): Linear(in_features=55, out_features=1024, bias=True)
    (1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): Dropout(p=0.4, inplace=False)
    (3): ReLU()
    (4): Linear(in_features=1024, out_features=1024, bias=True)
    (5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout(p=0.4, inplace=False)
    (7): ReLU()
    (8): Linear(in_features=1024, out_features=512, bias=True)
    (9): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): Dropout(p=0.4, inplace=False)
    (11): ReLU()
    (12): Linear(in_features=512, out_features=256, bias=True)
    (13): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): Dropout(p=0.4, inplace=False)
    (15): ReLU()
    (16): Linear(in_features=256, out_features=128, bias=True)
    (17): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): Dropout(p=0.4, inplace=False)
    (19): ReLU()
    (20): Linear(in_features=128, out_features=3, bias=True)
  )
)
```

Confusion Matrix and Accuracy:

```
Confusion matrix, without normalization
[[  6   1   7]
 [  7 281 239]
 [ 42  45 371]]
```




```
[26] print("Accuracy so far: " + str(100*accuracy_score(test_y, predicted))+ "%" )

Accuracy so far: 65.86586586586587%
```

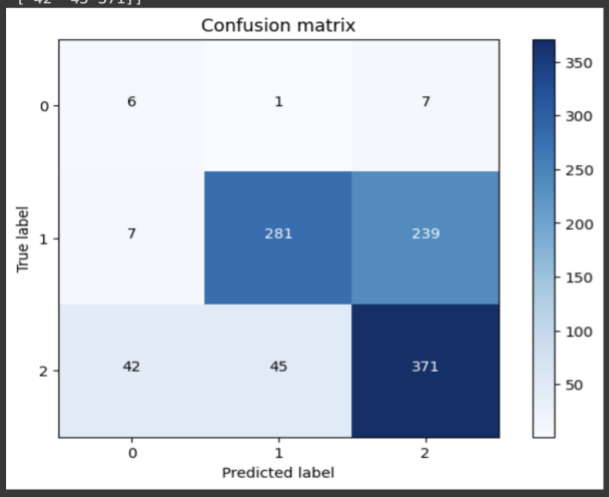
Adding Additional Metrics for the Model:

```
layers = [1024, 1024, 512, 256, 128]
model2 = Model(N, M,
               n_cont_user=df[user_specs].shape[1],
               n_cont_offer=df[offer_specs].shape[1],
               embed_dim=D,
               output_dim=df['event'].nunique(),
               layers=layers)

model2.load_state_dict(torch.load('/content/drive/MyDrive/ADSFinal/RecommendationModel2.pth'));
model2.to(device).eval()
```

```
Model(
  (bn_cont_u): BatchNorm1d(5, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn_cont_o): BatchNorm1d(10, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (u_emb): Embedding(16994, 20)
  (m_emb): Embedding(10, 20)
  (layers): Sequential(
    (0): Linear(in_features=55, out_features=1024, bias=True)
    (1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): Dropout(p=0.4, inplace=False)
    (3): ReLU()
    (4): Linear(in_features=1024, out_features=1024, bias=True)
    (5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout(p=0.4, inplace=False)
    (7): ReLU()
    (8): Linear(in_features=1024, out_features=512, bias=True)
    (9): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): Dropout(p=0.4, inplace=False)
    (11): ReLU()
    (12): Linear(in_features=512, out_features=256, bias=True)
    (13): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): Dropout(p=0.4, inplace=False)
    (15): ReLU()
    (16): Linear(in_features=256, out_features=128, bias=True)
    (17): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): Dropout(p=0.4, inplace=False)
    (19): ReLU()
    (20): Linear(in_features=128, out_features=3, bias=True)
```

Confusion Matrix and Accuracy:

```
Confusion matrix, without normalization
[[  6   1   7]
 [  7 281 239]
 [ 42  45 371]]
```




```
[38] print("Accuracy so far: " + str(100*accuracy_score(test_y, predicted))+ "%" )
```

```
Accuracy so far: 65.86586586586587%
```

F1 score for the model: 0.6660352920579345
Recall score for the model: 0.6586586586586587
Precision score for the model: 0.7305170743936796

Model Evaluation and Validation

An evaluation of multiple models with several parameters had been performed. It was practically established that using different optimizers, as well as tuning network depth and batch size can improve model accuracy. The most accurate models have the following results:

Accuracy

Best accuracy tested on randomly selected sample of data with 1000 rows.

Algorithm	Accuracy
Recommendation Engine 1	**68.36%**
Random Forest Model	**67.7%**
XGBoost Model	**70.7%**

With increasing the size of test data - accuracy grows by 3-5% for each model.

Accuracy of performance optimized models

Alternative approach that predict positive reaction or ignore action (negative) for an offer:

Algorithm	Accuracy	Precision
XGB Model	**98.0%**	0.96
Random Forest	**97.6%**	0.96

Metrics

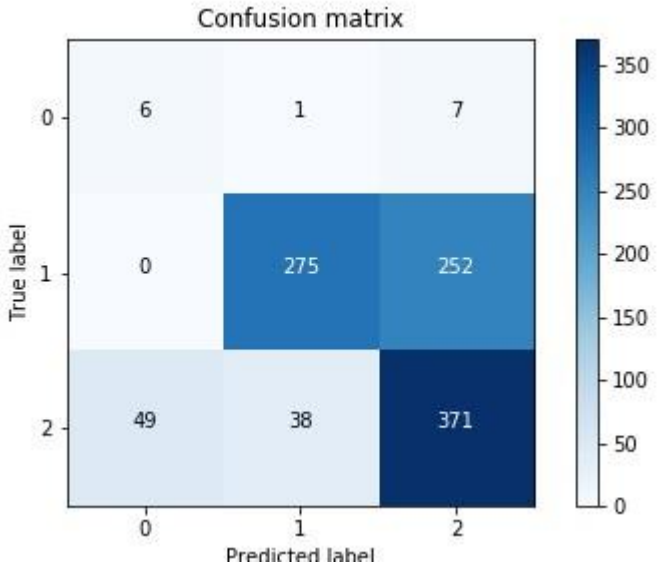
Other metrics are the following.

Algorithm	F1 Score	Recall	Precision
Recommendation Engine	0.66	0.67	0.73
Random Forest Model	0.674	0.67	0.67
Random Forest Model	0.695	0.7	0.69

Confusion Matrix

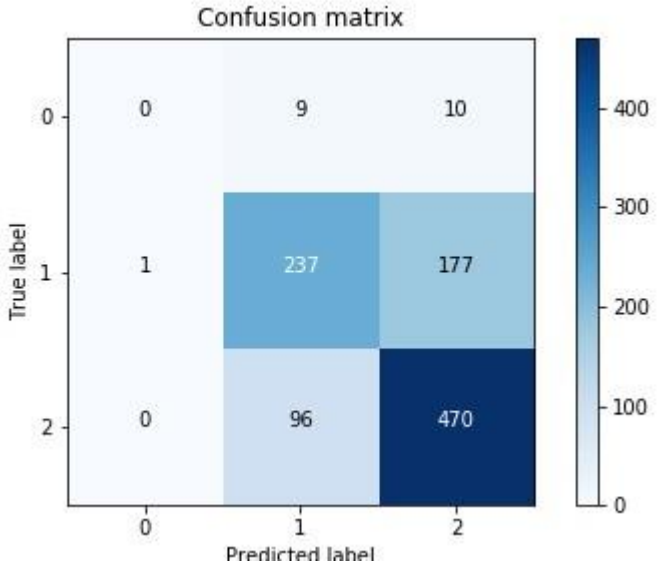
Recommendation Engine

For the classes [0,1,2] confusion matrix looks like the following:



Where 0 - offer ignored, 1 - offer viewed, 2 - offer completed.

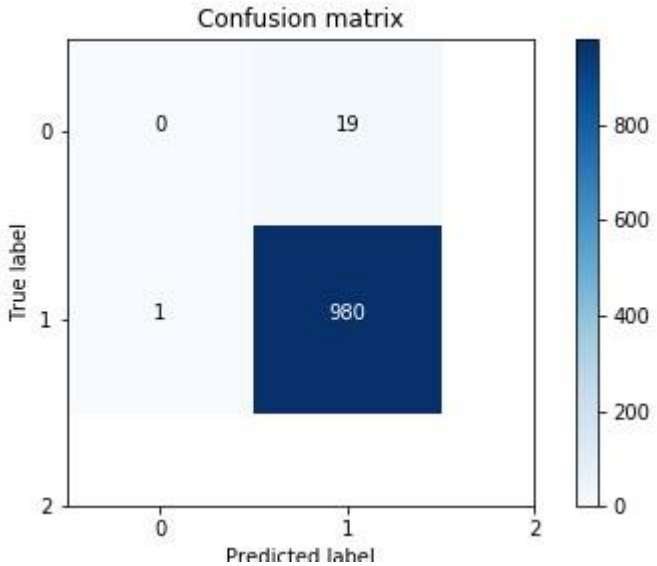
XGB Model



An alternative model can easily identify response/ignore outcome when binary value is used.

The model definitely can be used and can perform quick general prediction on large subsets of data.

Alternative Performance Optimized XGB Model



Conclusions:

This project is centered around developing a recommendation system for the Starbucks rewards mobile app.
1_StarbucksDataAnalysis: sets the stage with an analysis of customer behavior and the challenges of different offer types.
2_BaselineModel: tests a baseline model on the user data, focusing on understanding user actions such as ignoring, viewing, or completing offers.
3_RecommendationModel: progresses to a more refined recommendation model, likely leveraging more complex algorithms and data processing techniques to enhance the precision of the recommendations.
4_XGBoostModel: XGBoost model aimed at improving the prediction of customer responses to Starbucks' offers. The notebook includes model configuration, training processes, and performance metrics to assess the effectiveness of the XGBoost model in comparison to other models.

Acknowledgments:

The completion of this report was made possible through the collaboration and support of various individuals. I extend my gratitude to the paperswithcode for providing the dataset and references.