CONVOLVE 3.0 ROUND 2 SOLUTION REPORT

Approach: As per the problem statement, the task was to output all 28 time slots for each customer , ranked in decreasing order of their likelihood of opening an email during those slots. To address this, we chose to approach the problem as a **multi-class classification** task since each slot represents a distinct class.

This method allowed us to predict probabilities for all slots and rank them effectively, making the results easy to interpret.

EXPLORATORY DATA ANALYSIS

Data Loading and Initial Inspection

This project was initially provided with 2 datasets for training data, i.e. train_action_history.csv & train_cdna_data.csv .

1. Preprocessing Report for train_action_history.csv

The project commenced with loading the train_action_history dataset into a Pandas DataFrame for initial exploration and analysis.The data had **8797911 rows and 8 columns**.

The dataset had customer_code,send_timestamp,open_timestamp columns which were used to create the target column. We found that the number of unique customers in train_cdna were less than in train_action_history so we dropped all customer values that were not present in the train_cdna and in the train_action_history.

It was mentioned in the Problem Statement that to make sure that we join the cdna captured under the latest date and less than the send event in action history. So upon inspection of the train_cdna there were 6 batch dates that were found in the batch_date column. We mapped each of the send_timestamp to a corresponding batch date such that **send_timestamp<batch_date.**

We created a time_slot column by mapping each open_timestamp to one of 28 predefined time slots. If an email was not opened, it was assigned a value of 0, indicating no interaction within the corresponding time slot.

In one batch_date of the train_action_history there was only one instance of one customer. So every **combination of batch_date and customer_code is unique**.

To construct the target column for each unique combination of customer_code and batch_date we calculated the total number of emails opened in each of the 28 time slots.

The target column was then assigned the slot with the highest number of email openings. In cases where multiple slots had the highest count, the **median value** of these slots was selected as the target. This approach was chosen after evaluating the model's performance using both the median and the **first occurring highest value,** with the **median yielding better results.**

The three columns target_slot,batch_date and customer_code are kept in a different dataset that would be merged with the train_cdna dataset.

This final dataset had 3 columns and 647451 rows remaining.

2. Preprocessing Report for train_cdna.csv

Loading the train_cdna_data dataset into a Pandas DataFrame for initial exploration and analysis. The dataset consisted of **12,85,402 rows and 303 columns**.

This dataset had columns with numerical data, categorical data and also few columns with mixed data types.

The first step involved addressing two primary concerns:

1. **Handling Missing Values**: Several columns contained missing (NaN) values, which required appropriate cleaning strategies before training the model.
2. **Handling Categorical Data:** Columns with categorical or **mixed data types** had to be handled separately using one hot encoding.

Handling Missing Values

The first step in the preprocessing stage was to identify and handle missing values (NaNs) across all columns in the dataset. This was essential to ensure the integrity and accuracy of the data for model training.

Dropping Columns with Significant Missing Values

To address missing values in the dataset, we identified and removed columns **between 60-90%** missing values and those **over 95% missing values**.

Given the high proportion of missing data in these columns, retaining them would have added noise and complexity to the model without contributing meaningful information. Consequently, these columns were dropped from the dataset to reduce dimensionality.

Deducting that the missing values may also hold some significance, we encoded the columns having **90-95%** missing data replacing **NaNs with 0** and any other value with **1**.

Handling Individual Columns

- **Identifying columns with mixed data types & categorical columns that also had missing values:**

Column v2 : Contains age range. Replaced range with mean.

- `Columns with mixed data types (numerical + alphabetic + NaN values):`

```
['v5', 'v7', 'v9', 'v10', 'v15', 'v30', 'v31', 'v37', 'v63', 'v229',
'v230']
```

```
Total: 11
```

- `Columns with categorical data types that have NaN values:`

```
['v5', 'v6', 'v7', 'v9', 'v10', 'v11', 'v15', 'v27', 'v29', 'v30',
'v31', 'v33', 'v34', 'v35', 'v37', 'v42', 'v43', 'v54', 'v55', 'v56',
'v60', 'v63', 'v66', 'v68', 'v69', 'v71', 'v73', 'v74', 'v81', 'v99',
'v101', 'v102', 'v103', 'v229', 'v230', 'v271', 'v272', 'v273', 'v274',
'v275', 'v276', 'v277', 'v278', 'v279', 'v280', 'v281', 'v282', 'v283',
'v284', 'v285', 'v286']
```

```
Total: 51
```

Column v5: 1279548 entries were 99.0 clearly being the most frequent entry. A few ‘ZZ’ entries were present. Replaced the NaN values and ‘ZZ’ entries with 99.0

Column v7: Dropped v7 because it contained some alphanumeric values like 1159A, etc.

Columns v9, v10, v15, v30, v31, v37, 229, 230 : Dropped due to presence of too many categories. One hot encoding would not be helpful in this case.

Column v63: Contained Income range in string format. Mapped the range strings to numeric values (mean of range) for encoding. Filled missing values with median.

- **Identifying Numerical Columns with Missing Values**: The dataset included numerical columns, some of which had missing values. A thorough analysis was conducted using `isna().any()` on columns containing numerical data.

```
Those columns are: ['v24', 'v32', 'v44', 'v65', 'v77', 'v79', 'v80',
'v84', 'v128', 'v129', 'v130', 'v203', 'v204', 'v205', 'v206', 'v207',
'v208', 'v209', 'v224', 'v225', 'v226', 'v227', 'v228', 'v238', 'v239',
'v240', 'v241', 'v243', 'v244', 'v252', 'v253', 'v254', 'v255', 'v257',
'v258', 'v259', 'v260', 'v261', 'v262', 'v263', 'v264', 'v265', 'v266',
'v267', 'v268', 'v269', 'v270', 'age_mean']
```

- **Strategy for Handling Numerical Missing Values**: For numerical columns with missing values, the most suitable strategy was to replace the missing values with the **median** of each column. The median was chosen over the mean due to its robustness in handling outliers, which could skew the dataset. The median value for each column was calculated using the `median()` function in Pandas and used to fill the missing values (`fillna()` method). This method ensured that the distribution of data remained intact and minimized potential bias in the model.

- **Handling columns having categorical data with missing values:**

  ```
  Categorical columns with NaN values:

  ['v6', 'v11', 'v27', 'v29', 'v33', 'v34', 'v35', 'v42', 'v43',
  'v54', 'v55', 'v56', 'v60', 'v66', 'v68', 'v69', 'v71', 'v73',
  'v74', 'v81', 'v99', 'v101', 'v102', 'v103', 'v271', 'v272',
  'v273', 'v274', 'v275', 'v276', 'v277', 'v278', 'v279', 'v280',
  'v281', 'v282', 'v283', 'v284', 'v285', 'v286']



  Count of categorical columns with NaN values: 40
  ```

Dropped columns v6, v11, v27, v29, v33, v34, v37, v42, v56, v60, v66, v68, v69, v71, v73, v74, v81, v102, v103, v272, v273, v274, v275, v276, v277, v280, v281, v282, v283, v284, v285 due to reasons like data redundancy, too many categories which could not be encoded etc.

The rest of the columns that we decided to encode were v36 ,v43, v54, v55, v99, v101, v278, v286.

Before One Hot Encoding we **merged the train_cdna with the target_dataset** that was generated from the train_action_history because the number of columns in target_dataset is around 6.5 lakhs which is about **50% of the size of the current train_cdna**. This ensured faster operations by saving computational resources and helped make better decisions.

- **One Hot Encoding categorical data with missing values:** The columns were dealt with one by one systematically :
  1. The number of missing values in each categorical column was assessed to understand the extent of missing data.
  2. The unique values in each column were examined, along with their respective counts, to determine the distribution of categories.
  3. Based on the number of missing values and the distribution of categories, appropriate strategies were applied:
     - If the number of missing values was relatively low, they were imputed with the **most frequent value** in the column.

- If the number of missing values was significant, they were replaced with a generic category such as "Others" to prevent data sparsity and maintain model performance.

4. After handling missing values, categorical columns were processed using **one-hot encoding.** To prevent multicollinearity and adhere to the dummy variable trap principle, one category from each encoded feature was dropped.

MODEL DEVELOPMENT AND TRAINING

To develop an effective predictive model, we adopted a structured approach to training and tuning. Here, we outline the steps taken, along with the key numerical results obtained at each stage.

Data Preparation

We split the dataset into training and testing sets, allocating 80% of the data for training and 20% for testing.

- **Encoding Labels**:
  Since the XGBoost `num_classes` parameter, when set to 28, expects class labels ranging from **0 to 27**, the target column, which initially contained values from **1 to 28**, was transformed using **label encoding**. This conversion ensured compatibility with the model by shifting the class labels to the expected range, thus preventing potential indexing errors during training and evaluation.

- **Feature Normalization**:
  Features in the dataset, stored in `X`, were normalized using the **StandardScaler** to ensure that all variables contributed equally to the model training. This transformation standardized the features to have a mean of 0 and a standard deviation of 1.

Model Selection

Throughout our analysis, we explored a range of models to identify the best approach for our dataset. We experimented with popular boosting algorithms like **XGBoost**, recommendation systems such as **LightFM** and **Neural Collaborative Filtering (NCF)**, and more complex deep learning models like **Neural Networks**.

However, working with such a large dataset and within a limited timeframe posed significant challenges. While we gave LightFM and NCF a genuine try, their training processes couldn't be completed because of the heavy computational power they demanded.

In the end, we focused on comparing **XGBoost** and **Neural Networks**, both of which struck a balance between computational feasibility and performance.

XGBoost

Class Weight Calculation:
To address class imbalance, we calculated weights for each class using the `compute_class_weight` function. These weights were applied during training to ensure fair learning across all classes. But this was **not used** finally since it **did not improve model metrics.**

The XGBoost model was configured with the following parameters:

- **Objective:** `multi:softmax` to handle multi-class classification.
- **Number of Classes:** Set to 28, as per the problem's requirements.
- **Evaluation Metric:** Multi-class log-loss (`mlogloss`) for performance tracking.
- **Max Depth:** 10, to allow the model to capture complex patterns.
- **Number of Estimators:** 10, to balance training time and performance.

Model Training

- The model was trained on the scaled features (`X_train_scaled`) and encoded labels (`y_train_encoded`).

Predictions on Test Data

- **Class Predictions:**
  The model predicted the classes for the test dataset (`X_test_scaled`). These predictions were transformed back to their original labels using the label encoder.

Evaluation Metrics

- **Accuracy Score:**
  The accuracy of the model was evaluated using the `accuracy_score` function. This provided a measure of how well the predictions matched the true labels.

- **Mean Absolute Error (MAE):**
  The MAE was calculated to quantify the average prediction error, providing an additional perspective on model performance.

Sample Predictions

- A subset of 40 samples from the test dataset was selected for a detailed comparison of predicted and actual labels.
- Predictions for this subset were also transformed back to original labels for easier interpretability.

Prediction Probabilities

- **Probability Computation:**
  The model predicted class probabilities for the test dataset.
- **DataFrame Creation:**
  A DataFrame was created to store these probabilities, with each column representing the probability of a specific class.
- **Sorting Probabilities:**
  Probabilities for each row were sorted in descending order to identify the most likely classes.

Neural Network Architecture

- **Model Definition:**
  A sequential neural network was defined with the following layers:
  - Input Layer: Accepts input with dimensions equal to the number of features in `X2_train`.
  - Hidden Layers: Two dense layers with 128 and 64 neurons, respectively, each using ReLU activation and followed by a 30% dropout to prevent overfitting.
  - Output Layer: A softmax layer with 28 neurons, one for each class.
- **Compilation:**
  - Optimizer: Adam optimizer for efficient learning.
  - Loss Function: Categorical cross-entropy to handle multi-class classification.
  - Metric: Accuracy to evaluate performance.

Model Training

- The model was trained for 10 epochs with a batch size of 64, using the training set (`X2_train`, `y2_train`).
- Validation data (`X2_val`, `y2_val`) was used to monitor performance during training.

Evaluation and Predictions

- **Metrics:**
  - **Accuracy:** Measured the proportion of correct predictions.
  - **Mean Absolute Error (MAE):** Quantified the average prediction error.

Hyperparameter Tuning

During the parameter tuning phase of the neural network model, we evaluated multiple loss functions. However, none of these alternatives significantly improved the model's performance compared to **`categorical_crossentropy`, which we ultimately selected as the loss function.**

Additionally, we experimented with increasing the number of layers in the network architecture. However, this adjustment did not yield any notable improvement in performance, leading us to retain the original architecture.

We fine-tuned the Sequential model by adjusting:

- Number of neurons in input layer: 128.
- Dropout rates: 0.3 for the input layer and hidden layers.

Use of Mean Absolute Error (MAE) in Model Evaluation

In addition to standard classification metrics such as accuracy, we utilized **Mean Absolute Error (MAE)** to gain an alternative perspective on the model's performance. This choice was guided by the evaluation criteria, which specified the assessment of absolute distance from the true probability. MAE provided a straightforward and effective measure for comparing predicted probabilities to their true values.

The primary purpose of employing MAE was to measure the average magnitude of errors between the actual labels and the predicted labels.

Observations from MAE

For the **XGBoost**, the calculated MAE was **6.31**. After implementing the **Sequential neural network**, the MAE further increased to **7.85**

Since we did not have ground truth rankings we could not use Mean Average Precision as an evaluation metric.

TESTING DATA ANALYSIS AND PREPARATION

Testing Data Preparation

The test data was cleaned using the same steps as the training data, including handling missing values, encoding categorical variables, and aligning features. Columns were adjusted to ensure compatibility with the trained model.

Model Prediction on Test Data

The trained XGBoost model was applied to the test data (`X_test`) to predict the probabilities for each of the 28 email slots using the `predict_proba()` function. These probabilities were decoded using a label encoder and sorted in descending order to rank the slots for each customer. Finally, the rankings were added to the test dataset, and a CSV file was created for submission with the customer codes and their corresponding predicted slot rankings.

LIMITATIONS AND CHALLENGES FACED

We initially considered using LightFM and Neural Collaborative Filtering (NCF) for this task, as these models take into account **customer behaviors across all slots** rather than relying on a single target variable. They have the advantage of producing a **personalized ranking** of all slots, which aligns well with our goal of ranking slots based on customer interactions.

However, due to **computational limitations**, we were unable to train these models on such a **large dataset efficiently**. Additionally, a key challenge with LightFM was that it generates personalized rankings based on interactions **within the same dataset**. In our case, the train and test datasets had **completely different customer codes**, meaning the model trained on the train dataset could not be used directly for generating personalized rankings on the test data. This limitation highlighted the challenge of applying LightFM in a scenario where the test set comprises entirely new customers unseen during training.

COMPARISON & CONCLUSION

Both **XGBoost** and **Neural Networks** struggled to deliver strong performance in terms of accuracy; however, the **Mean Absolute Error (MAE)** showed a slight advantage for XGBoost over Neural Networks. Given the constraints that prevented us from utilizing advanced models like LightFM and NCF, we opted for XGBoost as the final model to predict on the test data.