DOCUMENTATION FOR THE MODEL PREPARED

Introduction

The primary goal of this model is to predict the most probable engagement time slots for customers based on their past communication and interaction data. Accurate time slot predictions are critical for optimizing email campaigns, ensuring that messages are sent when users are most likely to engage. This capability can significantly enhance customer experience and improve business outcomes by maximizing engagement rates and reducing resource wastage on ineffective communication strategies.

Objectives

1. Primary Objective: Identify the most likely time slots (out of 28 weekly slots) when a customer will engage with an email.
2. Secondary Objective: Ensure predictions are both scalable and robust, capable of handling large datasets efficiently while maintaining high accuracy.
3. Optimization Goals:
    - Improve engagement prediction accuracy by leveraging advanced machine learning techniques.
    - Employ preprocessing methods to ensure consistency across training and test datasets.
    - Address memory and computational limitations through efficient data handling strategies.

Importance of Time Slot Prediction

Efficient time slot prediction enables businesses to:

- Deliver personalized email campaigns tailored to user behavior.
- Maximize the likelihood of engagement by targeting optimal interaction windows.
- Reduce the cost associated with ineffective campaigns by focusing on high-probability slots.

Relevance to the Hackathon

This task aligns with the broader hackathon goal of solving real-world problems using data science and machine learning. By applying techniques such as feature engineering, LightGBM, and chunk-based data processing, this project showcases the practical application of machine learning to optimize marketing strategies.

The report documents the entire workflow, from preprocessing and feature generation to model training, prediction, and evaluation, highlighting the challenges faced and the solutions implemented to achieve robust results.

Data Overview

Dataset Description (For both the Test set and the Training Set)

The project utilized multiple datasets to ensure accurate predictions and robust model performance:

1. Communication History File:
    - Contains records of past email interactions, including timestamps of when emails were sent and opened.
    - Key columns include:
        - Customer_Code: Unique identifier for the customer.
        - Offer_id: Identifier for the offer.
        - Offer_subid: Sub-identifier for the offer (one offer_id can have multiple offer_subid values).
        - Product category and subcategory: Specifies the category and subcategory of the product tied to the offer.
        - Batch_id: Identifier for tracking events within the same communication batch.
        - Send_timestamp: Date and time when the email was sent.
        - Open_timestamp: Date and time when the email was opened. Null values indicate the email was not opened.
2. CDNA Dataset:
    - Includes customer-level features such as demographics and behavioral attributes.
    - Key considerations:
        - Captures data at regular intervals (generally weekly), stored in the batch_date column.
        - Features are aligned to ensure that only the latest available data prior to a communication event is used.
        - Not all features in the CDNA dataset are equally relevant; careful selection was necessary.
3. Submission File (this was available only for the test set)
    - A template file listing customer IDs for which engagement predictions are required.
    - Final predictions are appended to this file for submission.

Key Features

The dataset comprises: (for both test set and the training set)

- Timestamp Features:
    - send_timestamp and open_timestamp were processed to generate weekly engagement slots (28 slots).
    - Slots are defined based on 3-hour intervals, spanning Monday through Sunday.
- Categorical Features:
    - Variables like offer_id, product_category, and product_sub_category were encoded numerically to maintain consistency.

- Engagement Features:
    - Historical engagement metrics (slot_X_engagements) were utilized to capture prior customer interaction trends.

Data Preparation Steps

1. Decapitalization:
    - Ensured uniform column names across datasets to avoid mismatches during merging and processing.
2. Handling Missing Data:
    - LightGBM's native ability to handle missing values was leveraged, minimizing the need for imputation.
3. Feature Alignment:
    - Ensured that features in the training and test datasets matched exactly, including encoding categorical variables and renaming columns as required.
4. Chunk-Based Processing:
    - Test data was processed in manageable chunks to handle memory limitations effectively, ensuring scalability for large datasets.

METHODOLOGY

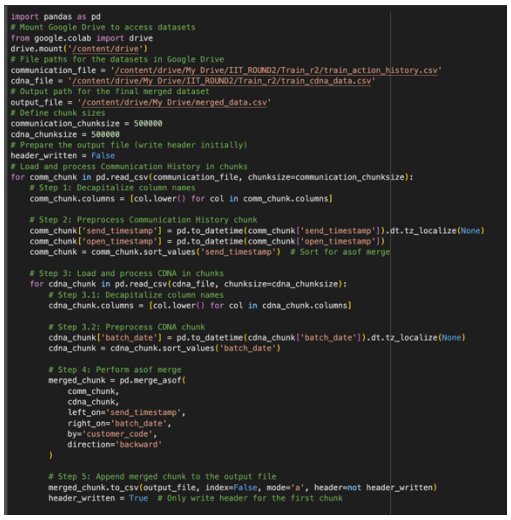
MERGING OF THE COMMUNICATION AND CDNA FILES FOR MODEL TRAINING

Merging Communication and CDNA Datasets

1. Purpose: To enrich the communication history with customer-level demographic and behavioural attributes from the CDNA dataset. This allows for a comprehensive feature set for training the engagement prediction model.
2. Steps:
    - Chunk-Based Processing:
        - Both the Communication and CDNA files were processed in chunks to handle memory constraints effectively.
    - Column Preprocessing:
        - Column names in both datasets were decapitalized to ensure consistency during the merge.
    - Timestamp Conversion:
        - send_timestamp and batch_date were converted to datetime objects and timezone information was removed for compatibility.
    - Sorting for Asof Merge:
        - The send_timestamp in the Communication file and batch_date in the CDNA file were sorted to facilitate the asof merge.
    - Asof Merge:
        - Performed an asof merge with customer_code as the grouping key, matching batch_date values in the CDNA dataset to the latest date prior to or equal to the send_timestamp in the Communication file.

- Output Handling:
  - Merged chunks were sequentially appended to an output CSV file to conserve memory.

CODE SNIPPET FOR LOADING AND MERGING OF TRAINING FILES



```
import pandas as pd
# Mount Google Drive to access datasets
from google.colab import drive
drive.mount('/content/drive')
# File paths for the datasets in Google Drive
communication_file = '/content/drive/My Drive/IIT_ROUND2/Train_r2/train_action_history.csv'
cdna_file = '/content/drive/My Drive/IIT_ROUND2/Train_r2/train_cdna_data.csv'
# Output path for the final merged dataset
output_file = '/content/drive/My Drive/merged_data.csv'
# Define chunk sizes
communication_chunksize = 500000
cdna_chunksize = 500000
# Prepare the output file (write header initially)
header_written = False
# Load and process Communication History in chunks
for comm_chunk in pd.read_csv(communication_file, chunksize=communication_chunksize):
    # Step 1: Decapitalize column names
    comm_chunk.columns = [col.lower() for col in comm_chunk.columns]

    # Step 2: Preprocess Communication History chunk
    comm_chunk['send_timestamp'] = pd.to_datetime(comm_chunk['send_timestamp']).dt.tz_localize(None)
    comm_chunk['open_timestamp'] = pd.to_datetime(comm_chunk['open_timestamp'])
    comm_chunk = comm_chunk.sort_values('send_timestamp')  # Sort for asof merge

    # Step 3: Load and process CDNA in chunks
    for cdna_chunk in pd.read_csv(cdna_file, chunksize=cdna_chunksize):
        # Step 3.1: Decapitalize column names
        cdna_chunk.columns = [col.lower() for col in cdna_chunk.columns]

        # Step 3.2: Preprocess CDNA chunk
        cdna_chunk['batch_date'] = pd.to_datetime(cdna_chunk['batch_date']).dt.tz_localize(None)
        cdna_chunk = cdna_chunk.sort_values('batch_date')

        # Step 4: Perform asof merge
        merged_chunk = pd.merge_asof(
            comm_chunk,
            cdna_chunk,
            left_on='send_timestamp',
            right_on='batch_date',
            by='customer_code',
            direction='backward'
        )

        # Step 5: Append merged chunk to the output file
        merged_chunk.to_csv(output_file, index=False, mode='a', header=not header_written)
        header_written = True  # Only write header for the first chunk
```

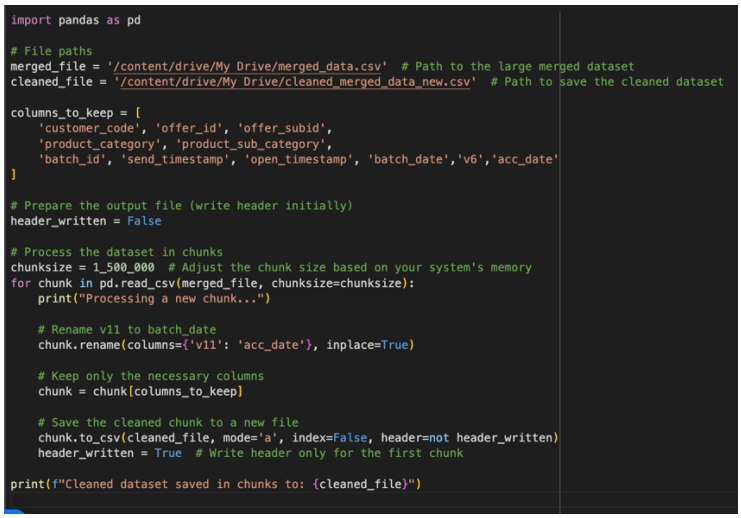
Now The Unnecessary Columns were dropped from the merged data

After merging the Communication and CDNA datasets, unnecessary columns were dropped to optimize the dataset for modeling. This step ensured that only relevant features were retained, reducing noise and improving model training efficiency.

Process:

1. Columns Retained:
   - Key columns included:
     - customer_code, offer_id, offer_subid, product_category, product_sub_category
     - batch_id, send_timestamp, open_timestamp, batch_date
     - Renamed column: v11 to acc_date
     - Additional feature: v6 (city names)
   - These columns were chosen for their potential predictive value based on domain knowledge.
2. Chunk-Based Processing:
   - The large merged dataset was processed in chunks of 1.5 million rows to handle memory constraints.
   - Each chunk was filtered to retain only the specified columns.
3. Output Handling:
   - The cleaned dataset was saved incrementally in a new CSV file, ensuring seamless handling of large data.

CODE SNIPPET FOR DROPPING THE COLUMNS



```python
import pandas as pd

# File paths
merged_file = '/content/drive/My Drive/merged_data.csv'  # Path to the large merged dataset
cleaned_file = '/content/drive/My Drive/cleaned_merged_data_new.csv'  # Path to save the cleaned dataset

columns_to_keep = [
    'customer_code', 'offer_id', 'offer_subid',
    'product_category', 'product_sub_category',
    'batch_id', 'send_timestamp', 'open_timestamp', 'batch_date','v6','acc_date'
]

# Prepare the output file (write header initially)
header_written = False

# Process the dataset in chunks
chunksize = 1_500_000  # Adjust the chunk size based on your system's memory
for chunk in pd.read_csv(merged_file, chunksize=chunksize):
    print("Processing a new chunk...")

    # Rename v11 to batch_date
    chunk.rename(columns={'v11': 'acc_date'}, inplace=True)

    # Keep only the necessary columns
    chunk = chunk[columns_to_keep]

    # Save the cleaned chunk to a new file
    chunk.to_csv(cleaned_file, mode='a', index=False, header=not header_written)
    header_written = True  # Write header only for the first chunk

print(f"Cleaned dataset saved in chunks to: {cleaned_file}")
```

Handling Mixed Data Types

To ensure consistency across the dataset and prepare it for model training, mixed datatypes were handled by assigning appropriate values. This step addressed potential discrepancies in column types that could hinder model training or evaluation.

Process:

1. String Columns:
    - Columns such as customer_code, offer_id, offer_subid, product_category, product_sub_category, and v6 (city names) were explicitly converted to string types to standardize their format.
2. Numeric Columns:
    - The batch_id column was converted to numeric using pd.to_numeric, with any non-numeric values coerced to NaN.
3. Datetime Columns:
    - Columns such as send_timestamp, open_timestamp, batch_date, and acc_date were converted to datetime objects using pd.to_datetime, with invalid entries coerced to NaT.
4. Output Handling:
    - The processed dataset was saved as a new CSV file to ensure data integrity and compatibility for downstream tasks.

CODE SNIPPET FOR HANDLING THE MIXED DATATYPES

```
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
# Path to the dataset
dataset_file = '/content/drive/My Drive/cleaned_merged_data_new.csv'

# Load the dataset
data = pd.read_csv(dataset_file)

# Handle string columns
string_columns = ['customer_code', 'offer_id', 'offer_subid', 'product_category', 'product_sub_category','v6']
for col in string_columns:
    data[col] = data[col].astype(str)

# Handle numeric columns
data['batch_id'] = pd.to_numeric(data['batch_id'], errors='coerce')

# Handle datetime columns
datetime_columns = ['send_timestamp', 'open_timestamp', 'batch_date','acc_date']
for col in datetime_columns:
    data[col] = pd.to_datetime(data[col], errors='coerce')

# Save the updated dataset
final_dataset_file = '/content/drive/My Drive/final_cleaned_processed_data_newedit.csv'
data.to_csv(final_dataset_file, index=False)

print(f"Final processed dataset saved to: {final_dataset_file}")
```

Feature Engineering and Target Variable Creation

This step involved generating new features and preparing the target variable to train the engagement prediction model effectively. The following steps were implemented:

Process:

1. Timestamp Conversion:
    - send_timestamp and open_timestamp were converted to datetime objects to facilitate time-based operations.
2. Weekly Time Slot Generation:
    - A custom function mapped timestamps to one of 28 weekly slots, defined as 3-hour intervals spanning Monday through Sunday:
        - Slot mapping: Monday Slot 1 = 1, Sunday Slot 28 = 28.
        - Non-engagement timestamps were marked as No Engagement.
3. Cleaning Slot Data:
    - "No Engagement" values were replaced with NaN, and slots were converted to numeric for downstream processing.
4. Target Variable Creation:
    - The binary target variable, same_slot_engagement, was created by checking if the send_time_slot matched the open_time_slot.
5. Historical Feature Aggregation:
    - For each customer, the number of engagements in each slot was counted:
        - Features were created as slot_X_engagements, where X corresponds to the slot number (1–28).
    - These features were merged back into the main dataset.
6. Output Handling:
    - The enriched dataset, including the target variable and historical engagement features, was saved for model training.

CODE SNIPPET FOR THE ABOVE

```
import pandas as pd
import numpy as np

# Path to the dataset
dataset_file = '/content/drive/My Drive/final_cleaned_processed_data_newedit.csv'

# Load the dataset
data = pd.read_csv(dataset_file)

# Step 1: Convert timestamps to datetime
data['send_timestamp'] = pd.to_datetime(data['send_timestamp'], errors='coerce')
data['open_timestamp'] = pd.to_datetime(data['open_timestamp'], errors='coerce')

# Step 2: Generate weekly time slots based on 3-hour intervals
def get_weekly_slot(timestamp):
    if pd.isna(timestamp):
        return "No Engagement"
    day_of_week = timestamp.dayofweek  # 0=Monday, 6=Sunday
    hour = timestamp.hour

    # Map 3-hour intervals to slots
    if 9 <= hour < 12:
        slot = 1
    elif 12 <= hour < 15:
        slot = 2
    elif 15 <= hour < 18:
        slot = 3
    elif 18 <= hour < 21:
        slot = 4
    else:
        return "No Engagement"

    # Calculate the slot number for the week
    return day_of_week * 4 + slot  # 1-28 slots (Monday slot_1 = 1, Sunday slot_28 = 28)

# Apply to send and open timestamps
data['send_time_slot'] = data['send_timestamp'].apply(get_weekly_slot)
data['open_time_slot'] = data['open_timestamp'].apply(get_weekly_slot)

# Step 3: Replace "No Engagement" with NaN and convert to numeric
data['send_time_slot'] = data['send_time_slot'].replace("No Engagement", np.nan)
data['open_time_slot'] = data['open_time_slot'].replace("No Engagement", np.nan)

# Convert time slots to numeric
data['send_time_slot'] = pd.to_numeric(data['send_time_slot'], errors='coerce')
data['open_time_slot'] = pd.to_numeric(data['open_time_slot'], errors='coerce')

# Step 4: Create the target variable (Same-Slot Engagement)
data['same_slot_engagement'] = (data['send_time_slot'] == data['open_time_slot']).astype(int)

# Step 5: Aggregate historical data for independent variables
# Count the number of times a customer engaged in each slot
historical_features = data.groupby(['customer_code', 'open_time_slot']).size().unstack(fill_value=0)
historical_features.columns = [f'slot_{int(col)}_engagements' for col in historical_features.columns]

# Merge historical features back into the dataset
data = data.merge(historical_features, how='left', left_on='customer_code', right_index=True)

# Step 6: Save the dataset with the target variable and features
final_dataset_with_target = '/content/drive/My Drive/dataset_with_same_slot_target_newedit.csv'
data.to_csv(final_dataset_with_target, index=False)

print(f"Dataset with same-slot target and historical features saved to: {final_dataset_with_target}")
```

Model Selection:

We are going to use boosting for this specific task because this technique handles tabular data very well.

What is Boosting?

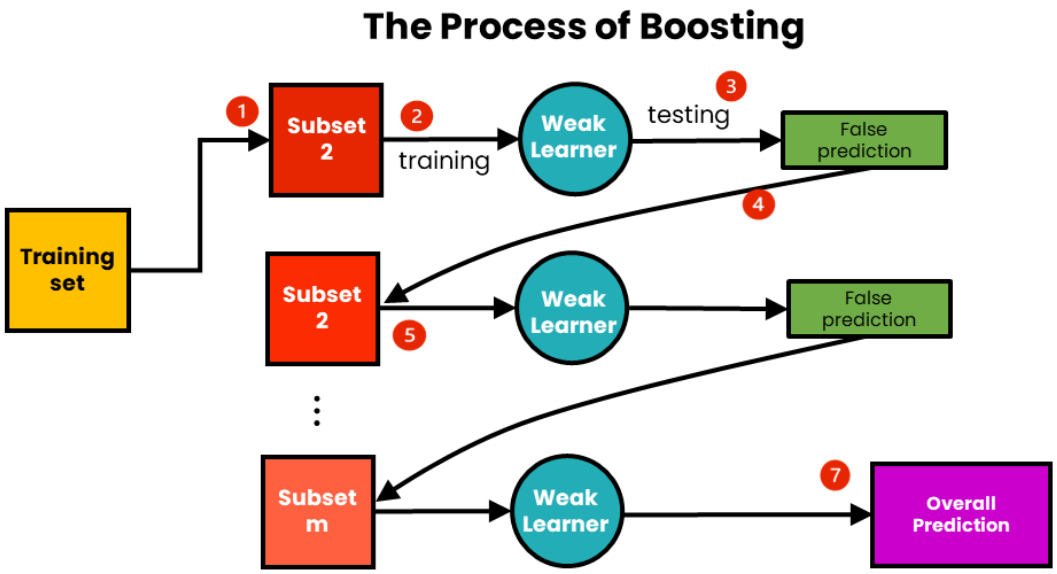
Boosting is a machine learning ensemble technique aimed at improving the accuracy of models by combining multiple weak learners (typically decision trees) to create a strong learner. Each weak learner performs slightly better than random guessing, and boosting enhances their collective performance by focusing on the mistakes of the previous learners.

Key Characteristics of Boosting:

1. Sequential Learning:
    - Boosting builds models sequentially, where each subsequent model focuses on correcting the errors made by the previous models.
2. Weighted Contribution:
    - Predictions from each model are weighted based on their performance. Better-performing models have higher weights in the final prediction.
3. Focus on Errors:
    - Misclassified instances or high-error regions are given higher weights or attention during the training of subsequent models.
4. Reduce Bias:
    - Boosting reduces both bias and variance, leading to improved generalization on unseen data.

The Process of Boosting

1. Initialize:
    - Start with a simple model, such as a decision tree with shallow depth.
2. Iterative Learning:
    - Train subsequent models to focus on the errors of the prior models. For example:
        - Misclassified samples are given higher weights for the next model.
        - Correctly classified samples are de-emphasized.
3. Aggregate Results:
    - Combine the predictions of all models using a weighted sum (or majority voting for classification) to make the final prediction.



Strengths of Boosting

- Can handle non-linear relationships effectively: Boosting is capable of modeling complex, non-linear patterns in the data.
- Reduces both bias and variance: By sequentially improving weak learners, boosting achieves a balance between bias and variance, leading to better generalization.
- Works well with structured and tabular data: Boosting techniques, like LightGBM, are particularly effective on tabular datasets.
- High accuracy in classification and regression tasks: Boosting often outperforms other algorithms in predictive tasks.

Limitations of Boosting

- Sensitive to noisy data and outliers: Boosting can magnify errors from noisy data or outliers, leading to suboptimal performance.
- Can lead to overfitting if not tuned properly: Without careful regularization or parameter tuning, boosting may overfit the training data.
- Computationally intensive for large datasets: Training boosting models can require significant time and computational resources without optimizations.

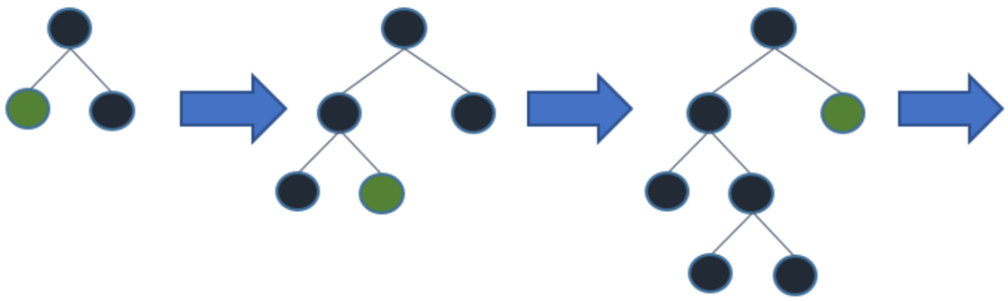
LightGBM (Light Gradient Boosting Machine) is an advanced implementation of the gradient boosting framework. It is optimized for efficiency, scalability, and high performance, making it one of the fastest and most accurate boosting methods available. Developed by Microsoft, LightGBM is designed to handle large-scale data and high-dimensional feature spaces efficiently.

Key Features of LightGBM

1. Leaf-Wise Tree Growth:
    - Traditional boosting methods grow decision trees level-wise (each level is fully grown before moving to the next).
    - LightGBM grows trees leaf-wise, splitting the leaf with the highest loss reduction. This leads to more accurate trees but might increase overfitting for small datasets.
2. Histogram-Based Splitting:
    - Continuous feature values are bucketed into discrete bins, significantly reducing computation time and memory usage.
    - This makes LightGBM faster, especially with high-dimensional data.
3. Categorical Feature Handling:
    - LightGBM can handle categorical features directly without requiring one-hot encoding.
    - It uses techniques like optimal split points for categorical variables, leading to better performance and faster training.
4. Efficient Memory Usage:
    - Designed to use memory efficiently by leveraging histogram-based computations and sparse data optimization.
5. Parallel Learning:
    - LightGBM supports parallel and distributed training, making it suitable for large datasets.
6. GPU Acceleration:
    - Training can be accelerated using GPUs, significantly reducing training time.

How LightGBM Works:

1. Initialization:
    - LightGBM initializes with a base prediction (e.g., the mean for regression or the class distribution for classification).
2. Iteration:
    - In each iteration, LightGBM computes the residuals (errors) from the previous predictions.
    - A decision tree is trained to minimize the residuals, using a loss function like mean squared error for regression or log-loss for classification.
    - Leaf-wise splitting is used to grow the tree.
3. Model Update:
    - The predictions are updated by adding the weighted contribution of the new tree to the overall model.
4. Repeat:
    - The process repeats for a specified number of iterations or until the loss function converges.



Leaf-wise tree growth

Advantages of Using LightGBM for This Project

1. Efficient Handling of Large-Scale Data:
    - LightGBM is optimized for speed and memory efficiency, making it ideal for handling large datasets like the merged communication and CDNA data.
2. Built-In Support for Categorical Features:
    - Unlike many algorithms, LightGBM can handle categorical data natively without requiring one-hot encoding, saving memory and reducing training time.
3. Ranking Capability:
    - The lambdarank objective is specifically designed for ranking tasks, making LightGBM an excellent choice for predicting engagement time slots where the goal is to rank slots by engagement probability.
4. Histogram-Based Splitting:
    - Continuous feature values are bucketed into discrete bins, significantly reducing computation time and memory usage, especially for high-dimensional data.

5. Regularization to Prevent Overfitting:
    - LightGBM provides L1L_1 and L2L_2 regularization parameters, which help control overfitting, ensuring the model generalizes well to unseen data.
6. Support for Missing Values:
    - LightGBM natively handles missing values, eliminating the need for imputation during preprocessing and making the workflow simpler.
7. Parallel and GPU Acceleration:
    - LightGBM supports parallel training and GPU acceleration, significantly reducing the training time for large datasets.
8. Customizable Objective Functions:
    - The flexibility to use custom objectives, like lambdarank for ranking, allows for better alignment with project-specific goals.
9. Feature Importance Analysis:
    - LightGBM provides built-in tools for analyzing feature importance, helping identify the most influential factors in predicting engagement time slots.
10. Scalability:
    - The ability to process data in chunks and handle datasets with millions of rows makes LightGBM scalable and well-suited for this project.

Parameter Tuning for the LightGBM Model

Objective of Parameter Tuning:

- To balance the trade-offs between overfitting and underfitting in the engagement time slot ranking task.
- To optimize the Mean Average Precision (MAP) metric, ensuring the model ranks engagement slots accurately.
- To enhance robustness and generalization across Group K-Fold validation splits, avoiding overfitting to specific groups or folds.

Parameter	Value	Purpose
objective	lambdarank	Specifies the objective function to optimize ranking tasks effectively.
metric	map	Evaluates the model using the Mean Average Precision (MAP) metric.
boosting_type	gbdt	Uses Gradient Boosting Decision Trees for iterative model improvement.
device	cpu	Ensures the model runs on the CPU, allowing

		compatibility with any system.
learning_rate	0.03	Controls the step size for each iteration, balancing convergence and stability.
num_leaves	10	Limits the number of leaves per tree to control model complexity.
max_depth	5	Restricts the depth of each tree to prevent overfitting.
lambda_l1	2.0	Applies L1 regularization to encourage sparsity in feature weights.
lambda_l2	2.0	Applies L2 regularization to shrink feature weights uniformly and prevent overfitting.
min_child_samples	150	Specifies the minimum number of data points per leaf to prevent overfitting.
seed	42	Ensures reproducibility by setting a fixed random seed.

Key Approach to Parameter Tuning

1. Focus on Ranking Optimization:
    - Selected the lambdarank objective to prioritize ranking engagement slots effectively.
    - Tuned parameters to improve the Mean Average Precision (MAP) metric, ensuring accurate slot rankings.
2. Regularization:
    - Employed strong L1L_1 (lambda_l1) and L2L_2 (lambda_l2) regularization values of 2.0 to prevent overfitting, particularly due to high-dimensional input features and the use of categorical encodings.
3. Balancing Model Complexity:
    - Limited tree depth (max_depth = 5) and the number of leaves (num_leaves = 10) to reduce overfitting while capturing relevant patterns in the data.
    - Increased min_child_samples to 150, ensuring leaf nodes had sufficient data points for stable predictions.
4. Learning Rate Adjustment:
    - Set a conservative learning rate (learning_rate = 0.03) to improve model stability and convergence, especially given the complexity of the ranking task.
5. Validation Strategy:

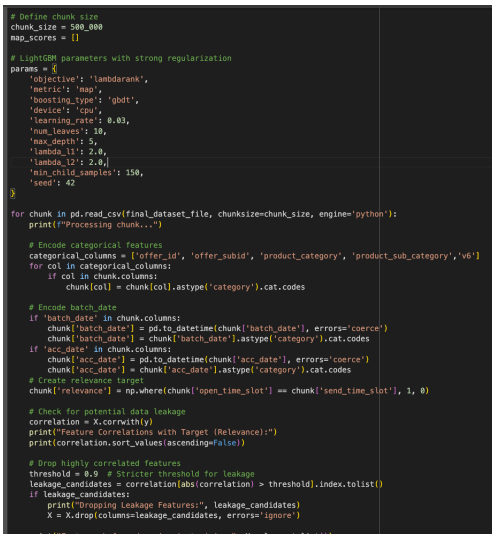
- Used Group K-Fold Cross-Validation with customer_code as the grouping key to assess performance across non-overlapping customer groups.
- Evaluated performance on each fold using the MAP metric to identify parameter settings that generalized well.

6. Iterative Tuning:
   - Conducted multiple rounds of parameter adjustments, analyzing MAP scores and overfitting trends to fine-tune key hyperparameters.

Observations from Parameter Tuning

1. Regularization Effect:
   - Strong L1L_1 and L2L_2 regularization effectively reduced overfitting, especially on highly correlated features and sparse categorical variables.
2. Impact of Learning Rate:
   - A lower learning rate of 0.03 provided better stability and allowed for finer updates to the model, which was critical for improving MAP scores across folds.
3. Tree Complexity:
   - Limiting tree depth (max_depth = 5) and the number of leaves (num_leaves = 10) struck the right balance between model complexity and generalization, reducing variance without sacrificing accuracy.
4. Min Child Samples:
   - Increasing min_child_samples to 150 improved generalization by preventing overly specific splits on small data groups.
5. MAP Scores Across Folds:
   - Consistently high MAP scores across folds demonstrated the robustness of the tuned parameters, with minimal variance between training and validation results.
6. Feature Contribution:
   - Regularization and correlation analysis highlighted key features contributing to the ranking task, enabling further refinement in feature engineering.

CODE SNIPPET FOR PARAMETER TUNING AND ENCODING



```
# Define chunk size
chunk_size = 500_000
map_scores = []

# LightGBM parameters with strong regularization
params = {
    'objective': 'lambdarank',
    'metric': 'map',
    'boosting_type': 'gbdt',
    'device': 'cpu',
    'learning_rate': 0.03,
    'num_leaves': 10,
    'max_depth': 5,
    'lambda_l1': 2.0,
    'lambda_l2': 2.0,
    'min_child_samples': 150,
    'seed': 42
}

for chunk in pd.read_csv(final_dataset_file, chunksize=chunk_size, engine='python'):
    print(f"Processing chunk...")

    # Encode categorical features
    categorical_columns = ['offer_id', 'offer_subid', 'product_category', 'product_sub_category','v6']
    for col in categorical_columns:
        if col in chunk.columns:
            chunk[col] = chunk[col].astype('category').cat.codes

    # Encode batch_date
    if 'batch_date' in chunk.columns:
        chunk['batch_date'] = pd.to_datetime(chunk['batch_date'], errors='coerce')
        chunk['batch_date'] = chunk['batch_date'].astype('category').cat.codes
    if 'acc_date' in chunk.columns:
        chunk['acc_date'] = pd.to_datetime(chunk['acc_date'], errors='coerce')
        chunk['acc_date'] = chunk['acc_date'].astype('category').cat.codes
    # Create relevance target
    chunk['relevance'] = np.where(chunk['open_time_slot'] == chunk['send_time_slot'], 1, 0)

    # Check for potential data leakage
    correlation = X.corrwith(y)
    print("Feature Correlations with Target (Relevance):")
    print(correlation.sort_values(ascending=False))

    # Drop highly correlated features
    threshold = 0.9  # Stricter threshold for leakage
    leakage_candidates = correlation[abs(correlation) > threshold].index.tolist()
    if leakage_candidates:
        print("Dropping Leakage Features:", leakage_candidates)
        X = X.drop(columns=leakage_candidates, errors='ignore')
```

Preventing Data Leakage and Handling Features

To ensure the integrity of the model and prevent data leakage, several steps were taken to identify and handle correlated features effectively:

1. Correlation Analysis:

- The correlation between each feature and the target variable (relevance) was calculated.
- Features with high absolute correlation (above a threshold of 0.9) were flagged as potential leakage candidates.

2. Identifying Leakage Features:

- Leakage features were identified based on their high correlation with the target, which might have inadvertently included information directly linked to the target variable.

3. Dropping Leakage Features:

- Identified leakage features were removed from the dataset to prevent the model from learning shortcuts or biases that could lead to artificially inflated evaluation metrics.

4. Feature Contribution Analysis:

- The mean relevance score was calculated for categorical features like v6 (city names) to assess their impact on engagement predictions.
- This analysis provided insights into the role of individual features, enabling informed decisions about their inclusion in the model.

5. Rechecking After Dropping Features:

- The dataset was re-evaluated after dropping leakage features to ensure the remaining features were free from high correlation with the target.

6. Adding Noise for Debugging:

- Noise (5% random changes) was introduced into the target variable (relevance) during training to simulate real-world imperfections and improve model robustness against noise

CODE SNIPPET FOR PREVENTING DATA LEAKAGE AND HANDLING FEATURES

```
# Create relevance target
chunk['relevance'] = np.where(chunk['open_time_slot'] == chunk['send_time_slot'], 1, 0)

# Check for potential data leakage
correlation = X.corrwith(y)
print("Feature Correlations with Target (Relevance):")
print(correlation.sort_values(ascending=False))

# Drop highly correlated features
threshold = 0.9  # Stricter threshold for leakage
leakage_candidates = correlation[abs(correlation) > threshold].index.tolist()
if leakage_candidates:
    print("Dropping Leakage Features:", leakage_candidates)
    X = X.drop(columns=leakage_candidates, errors='ignore')

print("Features before dropping in training:", X.columns.tolist())

# Prepare features and target
X = chunk.drop(columns=['relevance', 'send_timestamp', 'open_timestamp', 'customer_code',], errors='ignore')
y = chunk['relevance']
if 'v6' in chunk.columns:
    X['v6'] = chunk['v6']
# Check for potential data leakage
correlation = X.corrwith(y)
print("Feature Correlations with Target (Relevance):")
print(correlation.sort_values(ascending=False))
if 'v6' in chunk.columns:
   city_relevance = chunk.groupby('v6')['relevance'].mean()
print("Mean relevance by city (v6):")
print(city_relevance)
# Drop highly correlated features
threshold = 0.9  # Stricter threshold for leakage
leakage_candidates = correlation[abs(correlation) > threshold].index.tolist()
if leakage_candidates:
    print("Dropping Leakage Features:", leakage_candidates)
    X = X.drop(columns=leakage_candidates, errors='ignore')

# Add noise to the target for debugging
y = y.apply(lambda x: random.choice([0, 1]) if random.random() < 0.05 else x)
```

Group K-Fold Cross-Validation

What is Group K-Fold Cross-Validation?

Group K-Fold Cross-Validation is a validation technique used to assess the performance of machine learning models by splitting the dataset into kkk-folds, ensuring that data from the same group (e.g., customers in this project) does not appear in both the training and validation sets. The model is trained kkk times, with k−1k-1k−1 folds used as the training set and the remaining fold serving as the validation set during each iteration.

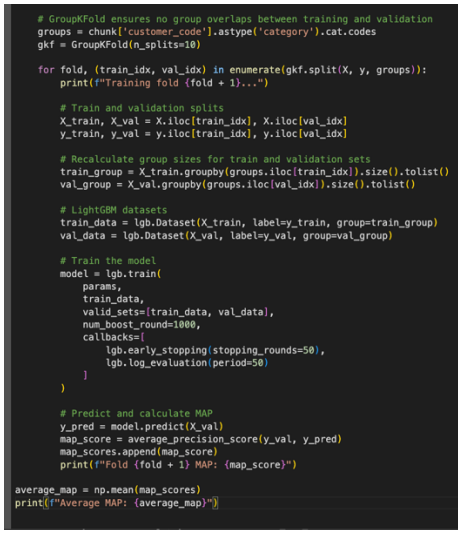
Performance metrics, such as Mean Average Precision (MAP), are calculated for each iteration and averaged across all kkk-folds to provide a comprehensive estimate of the model's effectiveness. This approach ensures all groups are evaluated, reduces overfitting to specific train-test splits, and provides a reliable measure of model performance compared to a single train-test split.

Why Group K-Fold Cross-Validation?

1. Grouped Data:
    - In this project, the dataset contains multiple interactions for each customer (customer_code), requiring careful handling to avoid data leakage between training and validation. Group K-Fold ensures customers do not appear in both sets, maintaining evaluation integrity.
2. Reliable Model Evaluation:
    - The task involves ranking engagement time slots for emails. Group K-Fold reduces bias and variance in performance estimates, improving generalization to unseen customer data.
3. Utilizing Limited Data Effectively:
    - Group K-Fold maximizes data usage by allowing every fold to contribute to both training and validation. This is particularly useful when dealing with large datasets chunked for memory constraints.
4. Generalizability:
    - By testing on completely unseen customer groups, Group K-Fold ensures the model generalizes well to new customers, aligning with the project's goal of accurate engagement predictions.

5. Robustness:
    - Using multiple train-test splits enhances robustness by minimizing the impact of fluctuations caused by random data partitioning. This ensures confidence in model predictions for real-world applications, such as optimizing email engagement.

CODE SNIPPET FOR GROUPK FOLD VALIDATION AND MODEL TRAINING
(WE LOAD THE MODEL IN CHUNKS AND PREDICTION IS MADE FOR THOSE CHUNKS ACROSS 10 FOLDS)



```
    # GroupKFold ensures no group overlaps between training and validation
    groups = chunk['customer_code'].astype('category').cat.codes
    gkf = GroupKFold(n_splits=10)

    for fold, (train_idx, val_idx) in enumerate(gkf.split(X, y, groups)):
        print(f"Training fold {fold + 1}...")

        # Train and validation splits
        X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
        y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

        # Recalculate group sizes for train and validation sets
        train_group = X_train.groupby(groups.iloc[train_idx]).size().tolist()
        val_group = X_val.groupby(groups.iloc[val_idx]).size().tolist()

        # LightGBM datasets
        train_data = lgb.Dataset(X_train, label=y_train, group=train_group)
        val_data = lgb.Dataset(X_val, label=y_val, group=val_group)

        # Train the model
        model = lgb.train(
            params,
            train_data,
            valid_sets=[train_data, val_data],
            num_boost_round=1000,
            callbacks=[
                lgb.early_stopping(stopping_rounds=50),
                lgb.log_evaluation(period=50)
            ]
        )

        # Predict and calculate MAP
        y_pred = model.predict(X_val)
        map_score = average_precision_score(y_val, y_pred)
        map_scores.append(map_score)
        print(f"Fold {fold + 1} MAP: {map_score}")

average_map = np.mean(map_scores)
print(f"Average MAP: {average_map}")
```

METRIC USED:

Mean Average Precision (MAP)

MAP (Mean Average Precision) is an evaluation metric widely used in ranking and classification tasks. It calculates the average precision across multiple queries, categories, or ranking lists, providing a single-figure measure of performance. MAP combines both precision and recall, making it an effective metric for scenarios where ranking quality matters.

How MAP Works:

- Precision is computed at each rank threshold, capturing the relevance of items retrieved up to that point.
- The average of precision scores across all relevant items is calculated for each query.
- Finally, the mean of these average precision scores across all queries provides the MAP score.

Why MAP Was Used in This Project

1. Focus on Ranking:
    - The primary goal of this project is to rank engagement time slots based on the probability of customer interaction. MAP evaluates the quality of the ranking, ensuring the most relevant slots appear higher in the list.

2. Accounts for Precision and Recall:
    - MAP provides a balanced evaluation by combining precision (how relevant the slots are) and recall (how many relevant slots are retrieved), ensuring an all-encompassing assessment of model performance.
3. Threshold-Free Evaluation:
    - Unlike accuracy or other metrics, MAP does not rely on a fixed decision threshold, making it ideal for ranking problems where relative ordering is more important.
4. Robust Metric for Real-World Scenarios:
    - MAP aligns with real-world requirements, such as prioritizing engagement slots, where correctly ranking the most relevant slots significantly impacts the outcome.

Results Of the model training

Model Performance:

- The LightGBM model was trained and validated in chunks due to the large size of the dataset.
- Group K-Fold Cross-Validation (10 folds) was performed independently on each chunk, ensuring no overlap of customer_code between training and validation splits.
- The MAP scores from each fold across all chunks were averaged to obtain the final performance metric:
    - Average MAP: 0.8422623710837985

Model saving to GDRIVE for future use:

```
import os

# Define the file path to save the model in Google Drive
model_path = '/content/drive/My Drive/final_lightgbm_model_newedit.txt'

# Save the LightGBM model
model.save_model(model_path)
print(f"Model saved to: {model_path}")
```

Test Dataset Prediction:

To prepare the test dataset for predictions, a similar methodology to the training dataset preprocessing was applied, focusing on dropping unnecessary columns and merging the communication and CDNA files efficiently.

1. Dropping Unnecessary Columns from CDNA

- Only the relevant columns were retained from the CDNA file to simplify the merging process:
    - customer_code: Unique identifier for each customer.
    - batch_date: Timestamp indicating when the CDNA data was captured.
    - v11: Batch-related data (renamed to acc_date in the training methodology).
    - v6: City name (useful for feature engineering and predictions).

- Dropping irrelevant columns helped reduce noise and memory usage, improving merge efficiency.

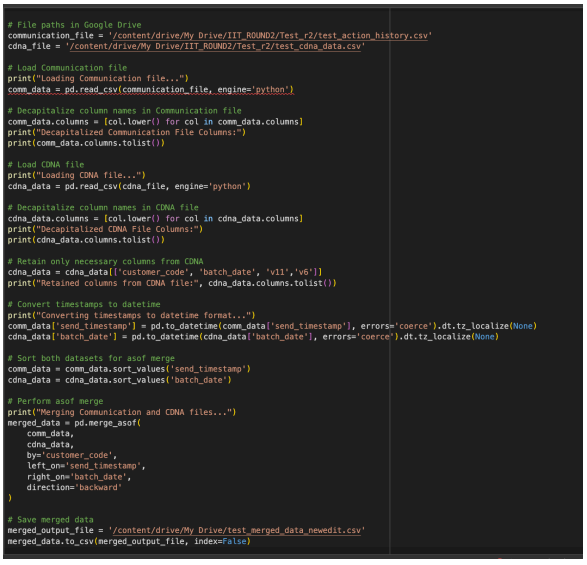
2. Merging Communication and CDNA Files

- The communication and CDNA files were merged using the following steps:
    - Decapitalization:
        - Column names in both datasets were converted to lowercase to ensure consistency.
    - Timestamp Conversion:
        - send_timestamp (from the communication file) and batch_date (from the CDNA file) were converted to datetime objects.
        - Timezone information was removed to align both columns.
    - Sorting:
        - Both datasets were sorted by their respective timestamp columns (send_timestamp and batch_date) to facilitate an asof merge.
    - Asof Merge:
        - Performed an asof merge on customer_code, matching the latest batch_date in the CDNA file that was earlier than or equal to the send_timestamp in the communication file.
        - This ensured that only relevant CDNA data was associated with each communication record.
    - Output Handling:
        - The merged dataset was saved to a new file for downstream prediction tasks.

Output:

- The final merged test dataset was saved as:
    - File Path: /content/drive/My Drive/test_merged_data_newedit.csv
- This dataset was prepared for further preprocessing and predictions.

CODE SNIPPET FOR MERGING OF CDNA AND COMMUNICATION TEST FILES



```
# File paths in Google Drive
communication_file = '/content/drive/My Drive/IIT_ROUND2/Test_r2/test_action_history.csv'
cdna_file = '/content/drive/My Drive/IIT_ROUND2/Test_r2/test_cdna_data.csv'

# Load Communication file
print("Loading Communication file...")
comm_data = pd.read_csv(communication_file, engine='python')

# Decapitalize column names in Communication file
comm_data.columns = [col.lower() for col in comm_data.columns]
print("Decapitalized Communication File Columns:")
print(comm_data.columns.tolist())

# Load CDNA file
print("Loading CDNA file...")
cdna_data = pd.read_csv(cdna_file, engine='python')

# Decapitalize column names in CDNA file
cdna_data.columns = [col.lower() for col in cdna_data.columns]
print("Decapitalized CDNA File Columns:")
print(cdna_data.columns.tolist())

# Retain only necessary columns from CDNA
cdna_data = cdna_data[['customer_code', 'batch_date', 'v11','v6']]
print("Retained columns from CDNA file:", cdna_data.columns.tolist())

# Convert timestamps to datetime
print("Converting timestamps to datetime format...")
comm_data['send_timestamp'] = pd.to_datetime(comm_data['send_timestamp'], errors='coerce').dt.tz_localize(None)
cdna_data['batch_date'] = pd.to_datetime(cdna_data['batch_date'], errors='coerce').dt.tz_localize(None)

# Sort both datasets for asof merge
comm_data = comm_data.sort_values('send_timestamp')
cdna_data = cdna_data.sort_values('batch_date')

# Perform asof merge
print("Merging Communication and CDNA files...")
merged_data = pd.merge_asof(
    comm_data,
    cdna_data,
    by='customer_code',
    left_on='send_timestamp',
    right_on='batch_date',
    direction='backward'
)

# Save merged data
merged_output_file = '/content/drive/My Drive/test_merged_data_newedit.csv'
merged_data.to_csv(merged_output_file, index=False)
```

TEST DATA PROCESSING FOR EVALUATION

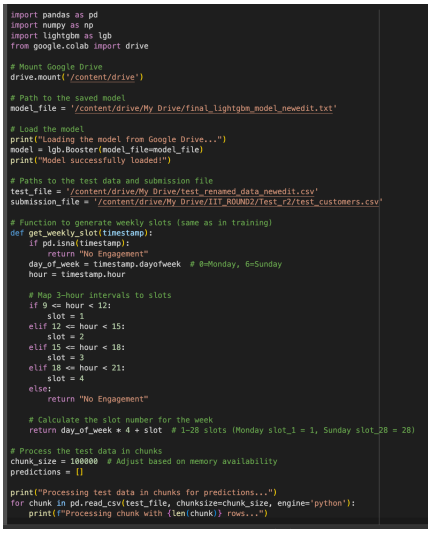
1. Test Data Preprocessing

- Objective: Prepare the test dataset for predictions by ensuring feature consistency with the training dataset and handling categorical, numerical, and datetime features appropriately.

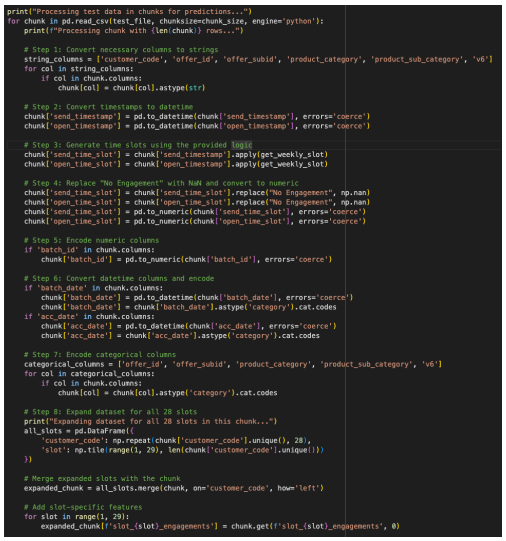
Steps:

1. Timestamp Conversion:
    - send_timestamp and open_timestamp were converted to datetime objects, allowing for time-based operations such as slot generation.
2. Weekly Slot Generation:
    - Using the same logic as the training process, weekly time slots (28 slots) were generated based on the send_timestamp and open_timestamp.
    - Slots represent 3-hour intervals over a week, from Monday (slot 1) to Sunday (slot 28).
3. Handling Missing Data:
    - "No Engagement" values were replaced with NaN for both send_time_slot and open_time_slot, ensuring these records were appropriately excluded during numeric operations.
4. Feature Encoding:
    - First the categorical columns were converted to strings to align the feature conversion followed in training.
    - Categorical columns (offer_id, product_category, v6, etc.) were then encoded using category codes to ensure compatibility with the LightGBM model.
    - Datetime columns (batch_date and acc_date) were also encoded as category codes.
5. Dataset Expansion:
    - Each customer was expanded to include all 28 possible weekly slots, ensuring that engagement probabilities could be predicted for all slots.
6. Slot-Specific Features:
    - Historical engagement features (slot_X_engagements) were integrated for each slot to align with the features used during training.

CODE SNIPPET FOR THE ABOVE STEPS



```
import pandas as pd
import numpy as np
import lightgbm as lgb
from google.colab import drive

# Mount Google Drive
drive.mount('/content/drive')

# Path to the saved model
model_file = '/content/drive/My Drive/final_lightgbm_model_newedit.txt'

# Load the model
print("Loading the model from Google Drive...")
model = lgb.Booster(model_file=model_file)
print("Model successfully loaded!")

# Paths to the test data and submission file
test_file = '/content/drive/My Drive/test_renamed_data_newedit.csv'
submission_file = '/content/drive/My Drive/IIT_ROUND2/Test_r2/test_customers.csv'

# Function to generate weekly slots (same as in training)
def get_weekly_slot(timestamp):
    if pd.isna(timestamp):
        return "No Engagement"
    day_of_week = timestamp.dayofweek  # 0=Monday, 6=Sunday
    hour = timestamp.hour

    # Map 3-hour intervals to slots
    if 9 <= hour < 12:
        slot = 1
    elif 12 <= hour < 15:
        slot = 2
    elif 15 <= hour < 18:
        slot = 3
    elif 18 <= hour < 21:
        slot = 4
    else:
        return "No Engagement"

    # Calculate the slot number for the week
    return day_of_week * 4 + slot  # 1-28 slots (Monday slot_1 = 1, Sunday slot_28 = 28)

# Process the test data in chunks
chunk_size = 100000  # Adjust based on memory availability
predictions = []

print("Processing test data in chunks for predictions...")
for chunk in pd.read_csv(test_file, chunksize=chunk_size, engine='python'):
    print(f"Processing chunk with {len(chunk)} rows...")
```



```
print("Processing test data in chunks for predictions...")
for chunk in pd.read_csv(test_file, chunksize=chunk_size, engine='python'):
    print(f"Processing chunk with {len(chunk)} rows...")

    # Step 1: Convert necessary columns to strings
    string_columns = ['customer_code', 'offer_id', 'offer_subid', 'product_category', 'product_sub_category', 'v6']
    for col in string_columns:
        if col in chunk.columns:
            chunk[col] = chunk[col].astype(str)

    # Step 2: Convert timestamps to datetime
    chunk['send_timestamp'] = pd.to_datetime(chunk['send_timestamp'], errors='coerce')
    chunk['open_timestamp'] = pd.to_datetime(chunk['open_timestamp'], errors='coerce')

    # Step 3: Generate time slots using the provided logic
    chunk['send_time_slot'] = chunk['send_timestamp'].apply(get_weekly_slot)
    chunk['open_time_slot'] = chunk['open_timestamp'].apply(get_weekly_slot)

    # Step 4: Replace "No Engagement" with NaN and convert to numeric
    chunk['send_time_slot'] = chunk['send_time_slot'].replace("No Engagement", np.nan)
    chunk['open_time_slot'] = chunk['open_time_slot'].replace("No Engagement", np.nan)
    chunk['send_time_slot'] = pd.to_numeric(chunk['send_time_slot'], errors='coerce')
    chunk['open_time_slot'] = pd.to_numeric(chunk['open_time_slot'], errors='coerce')

    # Step 5: Encode numeric columns
    if 'batch_id' in chunk.columns:
        chunk['batch_id'] = pd.to_numeric(chunk['batch_id'], errors='coerce')

    # Step 6: Convert datetime columns and encode
    if 'batch_date' in chunk.columns:
        chunk['batch_date'] = pd.to_datetime(chunk['batch_date'], errors='coerce')
        chunk['batch_date'] = chunk['batch_date'].astype('category').cat.codes
    if 'acc_date' in chunk.columns:
        chunk['acc_date'] = pd.to_datetime(chunk['acc_date'], errors='coerce')
        chunk['acc_date'] = chunk['acc_date'].astype('category').cat.codes

    # Step 7: Encode categorical columns
    categorical_columns = ['offer_id', 'offer_subid', 'product_category', 'product_sub_category', 'v6']
    for col in categorical_columns:
        if col in chunk.columns:
            chunk[col] = chunk[col].astype('category').cat.codes

    # Step 8: Expand dataset for all 28 slots
    print("Expanding dataset for all 28 slots in this chunk...")
    all_slots = pd.DataFrame({
        'customer_code': np.repeat(chunk['customer_code'].unique(), 28),
        'slot': np.tile(range(1, 29), len(chunk['customer_code'].unique()))
    })

    # Merge expanded slots with the chunk
    expanded_chunk = all_slots.merge(chunk, on='customer_code', how='left')

    # Add slot-specific features
    for slot in range(1, 29):
        expanded_chunk[f'slot_{slot}_engagements'] = chunk.get(f'slot_{slot}_engagements', 0)
```

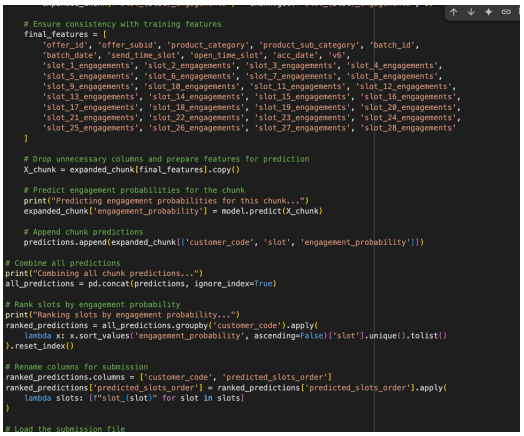
2. Prediction Process

- Objective: Predict engagement probabilities for each slot for all customers in the test dataset.

Steps:

1. Chunk-Based Processing:
    - The test dataset was processed in chunks of 100,000 rows to manage memory efficiently.
    - Predictions were generated independently for each chunk, and the results were combined after processing all chunks.
2. Model Prediction:
    - The pre-trained LightGBM model was loaded from Google Drive.
    - For each chunk, engagement probabilities for all 28 slots per customer were predicted using the model.
3. Ranking Engagement Slots:
    - For each customer, slots were ranked by their predicted engagement probabilities in descending order.
    - The ranked slots were formatted as slot_1, slot_2, etc., for submission.

CODE SNIPPET FOR MODEL PREDICTION



```
    # Ensure consistency with training features
    final_features = [
        'offer_id', 'offer_subid', 'product_category', 'product_sub_category', 'batch_id',
        'batch_date', 'send_time_slot', 'open_time_slot', 'acc_date', 'v6',
        'slot_1_engagements', 'slot_2_engagements', 'slot_3_engagements', 'slot_4_engagements',
        'slot_5_engagements', 'slot_6_engagements', 'slot_7_engagements', 'slot_8_engagements',
        'slot_9_engagements', 'slot_10_engagements', 'slot_11_engagements', 'slot_12_engagements',
        'slot_13_engagements', 'slot_14_engagements', 'slot_15_engagements', 'slot_16_engagements',
        'slot_17_engagements', 'slot_18_engagements', 'slot_19_engagements', 'slot_20_engagements',
        'slot_21_engagements', 'slot_22_engagements', 'slot_23_engagements', 'slot_24_engagements',
        'slot_25_engagements', 'slot_26_engagements', 'slot_27_engagements', 'slot_28_engagements'
    ]

    # Drop unnecessary columns and prepare features for prediction
    X_chunk = expanded_chunk[final_features].copy()

    # Predict engagement probabilities for the chunk
    print("Predicting engagement probabilities for this chunk...")
    expanded_chunk['engagement_probability'] = model.predict(X_chunk)

    # Append chunk predictions
    predictions.append(expanded_chunk[['customer_code', 'slot', 'engagement_probability']])

# Combine all predictions
print("Combining all chunk predictions...")
all_predictions = pd.concat(predictions, ignore_index=True)

# Rank slots by engagement probability
print("Ranking slots by engagement probability...")
ranked_predictions = all_predictions.groupby('customer_code').apply(
    lambda x: x.sort_values('engagement_probability', ascending=False)['slot'].unique().tolist()
).reset_index()

# Rename columns for submission
ranked_predictions.columns = ['customer_code', 'predicted_slots_order']
ranked_predictions['predicted_slots_order'] = ranked_predictions['predicted_slots_order'].apply(
    lambda slots: [f"slot_{slot}" for slot in slots]
)

# Load the submission file
```

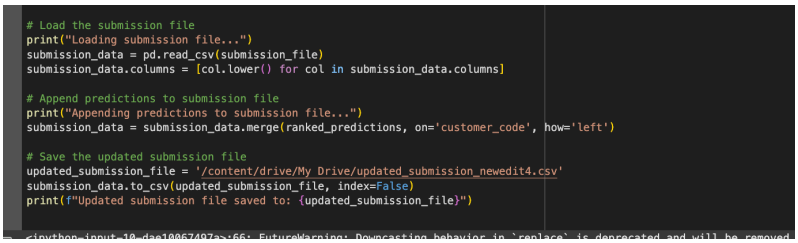
3. Submission File Preparation

- The predictions were appended to the provided submission file, which contained a list of customer codes.
- The final submission file included:
    - customer_code: Unique identifier for each customer.
    - predicted_slots_order: A ranked list of slots for each customer, based on engagement probabilities.

Output:

- The final submission file was saved as:
    - File Path: /content/drive/My Drive/updated_submission_newedit4.csv

CODE SNIPPET FOR APPENDING THE PREDICTIONS INTO THE SUBMISSION FILE AND SAVING IT AS A NEW FILE IN DRIVE



```
# Load the submission file
print("Loading submission file...")
submission_data = pd.read_csv(submission_file)
submission_data.columns = [col.lower() for col in submission_data.columns]

# Append predictions to submission file
print("Appending predictions to submission file...")
submission_data = submission_data.merge(ranked_predictions, on='customer_code', how='left')

# Save the updated submission file
updated_submission_file = '/content/drive/My Drive/updated_submission_newedit4.csv'
submission_data.to_csv(updated_submission_file, index=False)
print(f"Updated submission file saved to: {updated_submission_file}")
```

Key Observations:

1. Scalability:
    - The chunk-based processing ensured efficient handling of the large test dataset without exceeding memory limits.
2. Consistency:
    - Feature preparation and encoding matched the training methodology, ensuring compatibility with the pre-trained model.

3. Robust Ranking:
    - The predicted rankings reflected the model's ability to prioritize engagement slots effectively for each customer.

Conclusion

The project successfully addressed the challenge of predicting engagement time slots for customers based on historical communication and behavioral data. The model leveraged advanced feature engineering, chunk-based processing, and the LightGBM algorithm to make predictions for 28 weekly slots per customer. Predictions were based on engagement probabilities calculated for each slot, which were ranked in descending order to identify the most likely engagement windows for each customer.

Key Achievements:

1. High Model Performance:
    - The final LightGBM model achieved an Average MAP of 0.842, indicating effective ranking of engagement slots.
2. Scalability:
    - The methodology, including chunk-based processing and Group K-Fold validation, enabled seamless handling of large datasets while maintaining computational efficiency.

3. Feature Importance:
    - Incorporating historical engagement metrics and categorical features contributed significantly to the model's ability to prioritize engagement slots accurately.
4. Real-World Application:
    - The solution aligns with real-world marketing strategies, optimizing email campaigns and improving customer engagement.

Challenges:

While the project achieved its objectives, several challenges were encountered and addressed:

1. String Conversion and Encoding Consistency:
    - During the preprocessing of the test dataset, skipping string conversion for categorical columns initially caused inconsistent encodings between the training and test datasets.
    - This led to mismatched predictions for approximately 24% of customers due to incorrect feature mappings. The issue was resolved by ensuring all categorical columns were converted to strings before encoding.
2. Data Leakage and Correlation:
    - Some features showed high correlation with the target variable, risking data leakage. These were identified and removed to ensure the model’s predictions were unbiased.
3. Memory Constraints:

    - The large size of the dataset necessitated chunk-based processing for both training and prediction phases. Managing memory usage while maintaining data integrity was critical.

4. Imbalanced Data:
    - Certain slots had disproportionately low engagement rates. Feature engineering and robust evaluation metrics (MAP) were used to handle this imbalance effectively.
5. Model Generalization:
    - Ensuring that the model generalized well to unseen data required careful parameter tuning and the use of Group K-Fold validation to eliminate data overlap.
6. Time Complexity:
    - Expanding the dataset for all 28 slots per customer and processing each chunk individually increased computation time. Efficient feature encoding and optimization helped mitigate this challenge.

Technologies and Platforms Used

 Python:

- Used for data preprocessing, feature engineering, model training, prediction, and evaluation.
- Libraries like pandas, numpy, lightgbm, and sklearn .

 Google Colab:

- Used as the primary platform for running Python code, handling large datasets, and utilizing resources like GPU/TPU.
- Integrated with Google Drive for data storage and model saving.

 Google Drive:

- Used for storing datasets, models, and submission files.
- Enabled seamless data handling with Colab.

 LightGBM Framework:

- A machine learning platform specifically used for training and making predictions with the LightGBM model.

SUBMITTED BY
THE ERROR GUYS
(VISHESH GUPTA AND SOUHARDYO DASGUPTA)
(link to the notebook)
https://colab.research.google.com/drive/15lzfCImYaDqGB46POC7MHBnwDvpKIn4W?usp=sharing