```python
import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import roc_auc_score
import lightgbm as lgb
from tqdm import tqdm
import gc # Garbage collection
from datetime import datetime
import os


# sklearn.model_selection.StratifiedKFold: For stratified k-fold
cross-validation
# sklearn.preprocessing: Tools for data preprocessing (LabelEncoder,
StandardScaler)
# sklearn.metrics.roc_auc_score: For calculating area under ROC curve
# lightgbm (lgb): Gradient boosting framework optimized for efficiency
and performance
# tqdm: Provides progress bars for loops
# gc: Garbage collection for memory management
# datetime: For working with dates and times
# os: For interacting with the operating system

# Deep Learning Imports - PyTorch
# Ensure PyTorch is installed: pip install torch torchvision
torchaudio
try:
    import torch
    import torch.nn as nn
    import torch.optim as optim
    from torch.utils.data import DataLoader,Dataset
    PYTORCH_AVAILABLE = True
except ImportError:
    print("PyTorch not found. Deep Learning part will be skipped.
Install PyTorch (torch) to enable it.")
    PYTORCH_AVAILABLE = False

    # torch: Main PyTorch package providing tensor computations and
automatic differentiation
    # torch.nn: Neural network module containing layers, activation
functions, and loss functions
    # torch.optim: Package implementing various optimization
algorithms (SGD, Adam, etc.)
    # torch.utils.data.DataLoader: Utility for batch loading,
shuffling, and parallel data processing
    # torch.utils.data.TensorDataset: Simple dataset class wrapping
tensors
    # torch.utils.data.Dataset: Abstract class representing a dataset
for creating custom datasets
```

```python
# --- Configuration ---
DATA_PATH = '.' # <<< --- USER: PLEASE VERIFY THIS PATH ---
if not os.path.exists(DATA_PATH):
    print(f"ERROR: Data path '{DATA_PATH}' does not exist. Please
update the DATA_PATH variable.")

USER_LOG_FILE = os.path.join(DATA_PATH, 'user_log_format1.csv')
USER_INFO_FILE = os.path.join(DATA_PATH, 'user_info_format1.csv')
TRAIN_FILE = os.path.join(DATA_PATH, 'train_format1.csv')
TEST_FILE = os.path.join(DATA_PATH, 'test_format1.csv')
SUBMISSION_FILE = 'prediction_pytorch_lgbm.csv'
DL_MODEL_CHECKPOINT_PATH = 'best_dl_model_fold_{fold}.pth' # PyTorch
model extension

D11_MONTH = 11
D11_DAY = 11
D11_TIMESTAMP_INT = D11_MONTH * 100 + D11_DAY # 1111 represents 11/11

# --- Utility Functions ---
def convert_mmdd_to_days_before_d11(mmdd_series, ref_month=D11_MONTH,
ref_day=D11_DAY):
    days_in_month_cumulative = [0, 0, 31, 31+28, 31+28+31,
31+28+31+30, 31+28+31+30+31,
                               31+28+31+30+31+30,
31+28+31+30+31+30+31, 31+28+31+30+31+30+31+31,
                               31+28+31+30+31+30+31+31+30,
31+28+31+30+31+30+31+31+30+31,
                               31+28+31+30+31+30+31+31+30+31+30]

    def date_to_day_of_year(mmdd_str):
        if pd.isna(mmdd_str) or not isinstance(mmdd_str, str) or
len(mmdd_str) != 4:
            return np.nan
        try:
            m = int(mmdd_str[:2])
            d = int(mmdd_str[2:])
            if not (1 <= m <= 12 and 1 <= d <= 31): # Basic validation
                return np.nan
            return days_in_month_cumulative[m] + d
        except ValueError:
            return np.nan

    ref_day_of_year = date_to_day_of_year(f"{ref_month:02d}
{ref_day:02d}")
    if pd.isna(ref_day_of_year):
        raise ValueError("Reference date (D11) is invalid.")

    day_of_year_series = mmdd_series.apply(date_to_day_of_year)
    return ref_day_of_year - day_of_year_series
```

```
    # Function: convert_mmdd_to_days_before_d11
    #
    # Converts dates from 'mmdd' string format to the number of days
before D11 (November 11)
    #
    # Parameters:
    # - mmdd_series: Series of strings in 'mmdd' format (e.g., '1101'
for November 1)
    # - ref_month: Reference month (default: D11_MONTH which is 11 for
November)
    # - ref_day: Reference day (default: D11_DAY which is 11)
    #
    # Returns:
    # - Series with the number of days between each date and the
reference date (D11)
    # - Positive values indicate dates before D11, negative values are
after D11
    # - NaN values for invalid date formats or dates
    #
    # Note: Uses day-of-year calculation based on a non-leap year
calendar

# --- Data Loading and Basic Preprocessing ---
def load_data():
    print("Loading data...")
    # Processing Steps:
    #   1. Loads all CSV files with optimized data types to reduce
memory usage
    user_log_dtypes = {'user_id': np.uint32, 'item_id': np.uint32,
                       'cat_id': np.uint16, 'seller_id': np.uint16,
                       'brand_id': str, 'time_stamp': str,
'action_type': np.uint8}
    user_info_dtypes = {'user_id': np.uint32, 'age_range': str,
'gender': str}
    train_dtypes = {'user_id': np.uint32, 'merchant_id': np.uint16,
'label': np.uint8}
    test_dtypes = {'user_id': np.uint32, 'merchant_id': np.uint16}

    try:
        user_log = pd.read_csv(USER_LOG_FILE, dtype=user_log_dtypes)
        user_info = pd.read_csv(USER_INFO_FILE,
dtype=user_info_dtypes)
        train_data = pd.read_csv(TRAIN_FILE, dtype=train_dtypes)
        test_data = pd.read_csv(TEST_FILE, dtype=test_dtypes)
    except FileNotFoundError as e:
        print(f"ERROR: File not found. {e}. Please check your
DATA_PATH ('{DATA_PATH}') and file names.")
        raise
    #   2. Adds a placeholder 'prob' column to test data for later
predictions
```

```
    test_data['prob'] = 0.0

    print("Preprocessing basic data...")
    #   3. Renames 'seller_id' to 'merchant_id' in user_log for
consistency
    user_log.rename(columns={'seller_id': 'merchant_id'},
inplace=True)
    #   4. Converts demographic variables (age_range, gender) to
numeric types
    user_info['age_range'] = pd.to_numeric(user_info['age_range'],
errors='coerce').fillna(0).astype(np.uint8)
    user_info['gender'] = pd.to_numeric(user_info['gender'],
errors='coerce').fillna(2).astype(np.uint8)
    #   5. Processes timestamps and calculates days relative to the
D11 event (Nov 11)
    user_log['time_stamp_int'] = user_log['time_stamp'].apply(
        lambda x: int(x) if pd.notna(x) and isinstance(x, str) and
x.isdigit() and len(x) == 4 else -1
    )
    user_log['days_before_d11'] =
convert_mmdd_to_days_before_d11(user_log['time_stamp'])
    #   6. Creates a flag for records occurring on D11 (the major
shopping day)
    user_log['is_d11'] = (user_log['time_stamp_int'] ==
D11_TIMESTAMP_INT).astype(np.uint8)
    #   7. Cleans and converts brand_id to numeric format
    user_log['brand_id'] = pd.to_numeric(user_log['brand_id'],
errors='coerce').fillna(0).astype(np.uint32)
    #   8. Sorts user logs chronologically for each user
    user_log.sort_values(by=['user_id', 'time_stamp_int'],
ascending=[True, True], inplace=True)

    print("Data loaded and basic preprocessing done.")
    return user_log, user_info, train_data, test_data

    # load_data() Function Explanation
    #
    # Purpose:
    #   Loads and performs initial preprocessing of all datasets
needed for the analysis.
    #
    # Data Files:
    #   - USER_LOG_FILE: Contains user interaction logs (clicks,
purchases, etc.)
    #   - USER_INFO_FILE: Contains demographic information about users
    #   - TRAIN_FILE: Contains training data pairs (user_id,
merchant_id) with purchase labels
    #   - TEST_FILE: Contains test data pairs (user_id, merchant_id)
for prediction
```

```
    #
    #
    # Returns:
    #   - user_log: Processed user activity log data
    #   - user_info: Processed user demographic data
    #   - train_data: Labeled user-merchant pairs for model training
    #   - test_data: User-merchant pairs for prediction

# --- Feature Engineering Functions ---
def engineer_user_features(user_log, user_info):
    """Engineers features at the user level."""
    print("Engineering user-level features...")
    # begin with user demographic data
    features = user_info.copy()
    features.rename(columns={'age_range':'u_age_range',
'gender':'u_gender'}, inplace=True)
    # those history of logs before D11
    log_hist = user_log[user_log['days_before_d11'] > 0].copy()
    # User calculations below
    agg_funcs_hist = {
        'item_id': ['count', 'nunique'], 'cat_id': ['nunique'],
'merchant_id': ['nunique'],
        'brand_id': ['nunique'], 'days_before_d11': ['nunique', 'max',
'min', 'mean', 'std'],
    }
    # Group by user_id and aggregate
    user_activity_stats_hist =
log_hist.groupby('user_id').agg(agg_funcs_hist)
    # Prefix the columns with 'u_hist_'
    user_activity_stats_hist.columns = ['u_hist_' +
'_'.join(col).strip() for col in
user_activity_stats_hist.columns.values]
    # Rename columns for clarity
    user_activity_stats_hist.rename(columns={
        'u_hist_item_id_count': 'u_hist_total_actions',
'u_hist_item_id_nunique': 'u_hist_n_distinct_items',
        'u_hist_cat_id_nunique': 'u_hist_n_distinct_categories',
'u_hist_merchant_id_nunique': 'u_hist_n_distinct_merchants',
        'u_hist_brand_id_nunique': 'u_hist_n_distinct_brands',
'u_hist_days_before_d11_nunique': 'u_hist_days_active',
        'u_hist_days_before_d11_max':
'u_hist_earliest_action_days_prior',
        'u_hist_days_before_d11_min':
'u_hist_latest_action_days_prior',
        'u_hist_days_before_d11_mean':
'u_hist_mean_action_days_prior',
        'u_hist_days_before_d11_std': 'u_hist_std_action_days_prior',
    }, inplace=True)
    # Merge the aggregated features with the main features DataFrame
    features = features.merge(user_activity_stats_hist.reset_index(),
```

```
on='user_id', how='left')
    # Historical Action Type Counts:
    action_type_counts_hist = log_hist.groupby(['user_id',
'action_type']).size().unstack(fill_value=0)
    action_type_counts_hist.columns =
[f'u_hist_action_type_{col}_count' for col in
action_type_counts_hist.columns]
    features = features.merge(action_type_counts_hist.reset_index(),
on='user_id', how='left')

    for act_type in [0, 1, 2, 3]:
        col_name = f'u_hist_action_type_{act_type}_count'
        if col_name not in features.columns: features[col_name] = 0
    # Historical Ratios:
    features['u_hist_purchase_to_click_ratio'] =
features['u_hist_action_type_2_count'] /
(features['u_hist_action_type_0_count'] + 1e-6)
    features['u_hist_purchase_to_cart_ratio'] =
features['u_hist_action_type_2_count'] /
(features['u_hist_action_type_1_count'] + 1e-6)
    features['u_hist_purchase_to_fav_ratio'] =
features['u_hist_action_type_2_count'] /
(features['u_hist_action_type_3_count'] + 1e-6)
    features['u_hist_cart_to_click_ratio'] =
features['u_hist_action_type_1_count'] /
(features['u_hist_action_type_0_count'] + 1e-6)

    log_d11 = user_log[user_log['is_d11'] == 1].copy()
    user_d11_activity_counts = log_d11.groupby('user_id').agg(
        u_d11_total_actions = ('item_id', 'count'),
u_d11_n_distinct_items = ('item_id', 'nunique'),
        u_d11_n_distinct_merchants = ('merchant_id', 'nunique'),
u_d11_n_distinct_cats = ('cat_id', 'nunique')
    ).reset_index()
    features = features.merge(user_d11_activity_counts, on='user_id',
how='left')
    # "Double 11" General Activity (User's overall activity on D11):
    action_type_counts_d11 = log_d11.groupby(['user_id',
'action_type']).size().unstack(fill_value=0)
    action_type_counts_d11.columns = [f'u_d11_action_type_{col}_count'
for col in action_type_counts_d11.columns]
    features = features.merge(action_type_counts_d11.reset_index(),
on='user_id', how='left')
    # Temporal Window Features (Historical): window actions before D11
    for days_window in [1, 3, 7, 15, 30, 60, 90, 180]:
        temp_log_window = log_hist[log_hist['days_before_d11'] <=
days_window]
        user_window_actions = temp_log_window.groupby('user_id')
['item_id'].count().reset_index(name=f'u_actions_last_{days_window}d_p
```

```
rior')
        features = features.merge(user_window_actions, on='user_id',
how='left')
        user_window_purchases =
temp_log_window[temp_log_window['action_type']==2].groupby('user_id')
['item_id'].count().reset_index(name=f'u_purchases_last_{days_window}d
_prior')
        features = features.merge(user_window_purchases, on='user_id',
how='left')
    return features.fillna(0)
# fill nan to 0 if possible
def engineer_merchant_features(user_log):
    print("Engineering merchant-level features...")
    #Historical Popularity (Before "Double 11"):

    #How many interactions the merchant received in total.
    #How many unique users interacted with them.
    #How many different items, brands, and categories they handled.
    #Counts of different action types (clicks, purchases, etc.)
directed at them.
    #Their historical conversion rate (purchases / clicks).
    #"Double 11" Activity:

    #How many interactions they received on "Double 11".
    #How many unique users interacted with them on "Double 11".
    #Counts of different action types they received on "Double 11".
    unique_merchants = user_log['merchant_id'].unique()
    features = pd.DataFrame({'merchant_id':
unique_merchants[pd.notna(unique_merchants)]})
    log_hist = user_log[user_log['days_before_d11'] > 0].copy()
    agg_funcs_m_hist = {'user_id': ['count', 'nunique'], 'item_id':
['nunique'], 'brand_id': ['nunique'], 'cat_id': ['nunique']}
    merchant_stats_hist =
log_hist.groupby('merchant_id').agg(agg_funcs_m_hist)
    merchant_stats_hist.columns = ['m_hist_' + '_'.join(col).strip()
for col in merchant_stats_hist.columns.values]
    merchant_stats_hist.rename(columns={
        'm_hist_user_id_count': 'm_hist_total_interactions_received',
'm_hist_user_id_nunique': 'm_hist_n_distinct_users',
        'm_hist_item_id_nunique': 'm_hist_n_distinct_items_handled',
'm_hist_brand_id_nunique': 'm_hist_n_distinct_brands_handled',
        'm_hist_cat_id_nunique':
'm_hist_n_distinct_categories_handled'}, inplace=True)
    features = features.merge(merchant_stats_hist.reset_index(),
on='merchant_id', how='left')

    merchant_action_counts_hist = log_hist.groupby(['merchant_id',
'action_type']).size().unstack(fill_value=0)
    merchant_action_counts_hist.columns =
[f'm_hist_action_type_{col}_count' for col in
```

```
merchant_action_counts_hist.columns]
    features =
features.merge(merchant_action_counts_hist.reset_index(),
on='merchant_id', how='left')
    for act_type in [0, 1, 2, 3]:
        col_name = f'm_hist_action_type_{act_type}_count'
        if col_name not in features.columns: features[col_name] = 0
    features['m_hist_conversion_rate'] =
features['m_hist_action_type_2_count'] /
(features['m_hist_action_type_0_count'] + 1e-6)

    log_d11 = user_log[user_log['is_d11'] == 1].copy()
    merchant_d11_activity_counts = log_d11.groupby('merchant_id').agg(
        m_d11_total_interactions = ('item_id', 'count'),
m_d11_n_distinct_users = ('user_id', 'nunique'),
        m_d11_n_distinct_items = ('item_id', 'nunique')).reset_index()
    features = features.merge(merchant_d11_activity_counts,
on='merchant_id', how='left')
    merchant_d11_action_counts = log_d11.groupby(['merchant_id',
'action_type']).size().unstack(fill_value=0)
    merchant_d11_action_counts.columns =
[f'm_d11_action_type_{col}_count' for col in
merchant_d11_action_counts.columns]
    features =
features.merge(merchant_d11_action_counts.reset_index(),
on='merchant_id', how='left')
    return features.fillna(0)

def engineer_user_merchant_interaction_features(user_log, base_df):
    print("Engineering user-merchant interaction features...")
    # user-merchant interaction features before D11
    log_hist = user_log[user_log['days_before_d11'] > 0].copy()
    um_interactions_hist_agg = log_hist.groupby(['user_id',
'merchant_id']).agg(
        um_hist_total_actions=('item_id', 'count'),
um_hist_distinct_items=('item_id', 'nunique'),
        um_hist_distinct_cats=('cat_id', 'nunique'),
um_hist_distinct_brands=('brand_id', 'nunique'),
        um_hist_last_interaction_days_prior=('days_before_d11',
'min'),
        um_hist_first_interaction_days_prior=('days_before_d11',
'max'),
        um_hist_days_active_with_merchant=('days_before_d11',
'nunique')).reset_index()
    merged_df = base_df.merge(um_interactions_hist_agg, on=['user_id',
'merchant_id'], how='left')

    um_action_counts_hist = log_hist.groupby(['user_id',
'merchant_id', 'action_type']).size().unstack(fill_value=0)
```

```
    um_action_counts_hist.columns =
[f'um_hist_action_type_{col}_count' for col in
um_action_counts_hist.columns]
    merged_df = merged_df.merge(um_action_counts_hist.reset_index(),
on=['user_id', 'merchant_id'], how='left')
    # user-merchant interaction features on D11
    log_d11 = user_log[user_log['is_d11'] == 1].copy()
    um_d11_interactions_agg = log_d11.groupby(['user_id',
'merchant_id']).agg(
        um_d11_total_actions=('item_id', 'count'),
um_d11_purchases=('action_type', lambda x: (x == 2).sum()),
        um_d11_clicks=('action_type', lambda x: (x == 0).sum()),
um_d11_carts=('action_type', lambda x: (x == 1).sum()),
        um_d11_favs=('action_type', lambda x: (x == 3).sum()),
        um_d11_distinct_items_interacted=('item_id',
'nunique')).reset_index()
    merged_df = merged_df.merge(um_d11_interactions_agg,
on=['user_id', 'merchant_id'], how='left')
    # more weight to recent interactions
    decay_rate = 0.01
    log_hist.loc[:, 'interaction_weight'] = np.exp(-decay_rate *
log_hist['days_before_d11'])
    um_time_decayed_score = log_hist.groupby(['user_id',
'merchant_id'])
['interaction_weight'].sum().reset_index(name='um_hist_time_decayed_sc
ore')
    merged_df = merged_df.merge(um_time_decayed_score, on=['user_id',
'merchant_id'], how='left')
    # first item the user purchased from this specific merchant on
"Double 11", what kind of product initiated the "new buyer"
relationship.
    d11_purchases = log_d11[log_d11['action_type'] == 2]
    first_d11_purchase_details =
d11_purchases.drop_duplicates(subset=['user_id', 'merchant_id'],
keep='first')
    acquisition_item_features = first_d11_purchase_details[['user_id',
'merchant_id', 'item_id', 'cat_id', 'brand_id']]
    acquisition_item_features.columns = ['user_id', 'merchant_id',
'acq_item_id', 'acq_cat_id', 'acq_brand_id']
    merged_df = merged_df.merge(acquisition_item_features,
on=['user_id', 'merchant_id'], how='left')
    # no prior interactions with a merchant, or no "Double 11"
interaction
    interaction_cols_to_fill = [col for col in merged_df.columns if
col.startswith('um_') or col.startswith('acq_')]
    for col in interaction_cols_to_fill:
        if 'days_prior' in col or 'score' in col:
merged_df[col].fillna(-1, inplace=True)
        elif col in ['acq_item_id', 'acq_cat_id', 'acq_brand_id']:
```

```
merged_df[col].fillna(0, inplace=True)
        else: merged_df[col].fillna(0, inplace=True)
    return merged_df

def create_all_features(user_log, user_info, train_data, test_data):
    print("Starting comprehensive feature engineering...")
    train_ids_df = train_data[['user_id', 'merchant_id']].copy()
    test_ids_df = test_data[['user_id', 'merchant_id']].copy()
    train_ids_df['_is_train_data_source'] = 1
    test_ids_df['_is_train_data_source'] = 0
    all_pairs_df = pd.concat([train_ids_df, test_ids_df],
axis=0).drop_duplicates(subset=['user_id', 'merchant_id'])
    # combine train and test targets
    df_user_features = engineer_user_features(user_log, user_info)
    all_pairs_featured_df = all_pairs_df.merge(df_user_features,
on='user_id', how='left')
    del df_user_features; gc.collect()
    # all general user characteristics
    df_merchant_features = engineer_merchant_features(user_log)
    all_pairs_featured_df =
all_pairs_featured_df.merge(df_merchant_features, on='merchant_id',
how='left')
    del df_merchant_features; gc.collect()
    # all general merchant characteristics
    all_pairs_featured_df =
engineer_user_merchant_interaction_features(user_log,
all_pairs_featured_df)
    # how specific user interacted with specific merchant
    train_featured_df =
all_pairs_featured_df[all_pairs_featured_df['_is_train_data_source']
== 1].drop(columns=['_is_train_data_source'])
    test_featured_df =
all_pairs_featured_df[all_pairs_featured_df['_is_train_data_source']
== 0].drop(columns=['_is_train_data_source'])

    train_featured_df = train_featured_df.merge(train_data[['user_id',
'merchant_id', 'label']], on=['user_id', 'merchant_id'], how='left')
    test_featured_df = test_data[['user_id', 'merchant_id',
'prob']].merge(test_featured_df.drop(columns=['prob'],
errors='ignore'), on=['user_id', 'merchant_id'], how='left')
    # Split the data into fully featured train and test sets
    ratio_feature_defs = [
        ('um_hist_total_actions', 'u_hist_total_actions',
'ratio_um_hist_actions_vs_u_hist_actions'),
        ('um_d11_total_actions', 'u_d11_total_actions',
'ratio_um_d11_actions_vs_u_d11_actions'),
        ('um_d11_purchases', 'u_d11_action_type_2_count',
'ratio_um_d11_purchases_vs_u_d11_purchases'),
        ('um_d11_total_actions', 'm_d11_total_interactions',
```

```
'ratio_um_d11_actions_vs_m_d11_interactions'),
    ]
    for df_iter in [train_featured_df, test_featured_df]:
        for num_col, den_col, ratio_col_name in ratio_feature_defs:
            if num_col in df_iter.columns and den_col in
df_iter.columns:
                df_iter[ratio_col_name] = df_iter[num_col] /
(df_iter[den_col] + 1e-6)
            else:
                df_iter[ratio_col_name] = 0
    train_featured_df.fillna(0, inplace=True)
    test_featured_df.fillna(0, inplace=True)
    # calculate the ratios and deal with the missing values
    print("Feature engineering complete.")
    return train_featured_df, test_featured_df

# --- Model Training (LightGBM) ---
def train_predict_lgbm(train_df, test_df, features_to_use,
target_col='label', n_splits=5):
    print("Training LightGBM model...")
    # Separate train and test data
    X = train_df[features_to_use].copy()
    y = train_df[target_col]
    X_test = test_df[features_to_use].copy()

    oof_preds = np.zeros(X.shape[0])
    test_preds = np.zeros(X_test.shape[0])
    # Convert categorical features to 'category' dtype for LightGBM
    categorical_feature_names = ['u_age_range', 'u_gender',
'merchant_id', 'acq_item_id', 'acq_cat_id', 'acq_brand_id']
    categorical_features_for_lgbm = [f for f in
categorical_feature_names if f in features_to_use]
    for col in categorical_features_for_lgbm:
        if col in X.columns: X.loc[:, col] = X[col].astype('category')
        if col in X_test.columns: X_test.loc[:, col] =
X_test[col].astype('category')
    # Stratified K-Fold cross-validation
    skf = StratifiedKFold(n_splits=n_splits, shuffle=True,
random_state=42)
    # Initialize LightGBM parameters
    params = {
        'objective': 'binary', 'metric': 'auc', 'boosting_type':
'gbdt',
        'n_estimators': 3000, 'learning_rate': 0.01, 'num_leaves': 42,
'max_depth': 7,
        'seed': 42, 'n_jobs': -1, 'verbose': -1, 'colsample_bytree':
0.7, 'subsample': 0.7,
        'subsample_freq': 1, 'reg_alpha': 0.15, 'reg_lambda': 0.15,
    }
```

```
    # DataFrame to store feature importances
    feature_importances_df = pd.DataFrame(index=features_to_use)
    # Training loop and predictions
    for fold, (train_idx, val_idx) in enumerate(tqdm(skf.split(X, y),
total=n_splits, desc="LGBM Folds")):
        X_train, y_train = X.iloc[train_idx], y.iloc[train_idx]
        X_val, y_val = X.iloc[val_idx], y.iloc[val_idx]
        model = lgb.LGBMClassifier(**params)
        model.fit(X_train, y_train, eval_set=[(X_val, y_val)],
eval_metric='auc',
                  callbacks=[lgb.early_stopping(150, verbose=False)],
                  categorical_feature=categorical_features_for_lgbm if
categorical_features_for_lgbm else 'auto')
        oof_preds[val_idx] = model.predict_proba(X_val)[:, 1]
        test_preds += model.predict_proba(X_test)[:, 1] / n_splits
        feature_importances_df[f'fold_{fold+1}'] =
pd.Series(model.feature_importances_, index=features_to_use)
    # report the OOF AUC performance
    oof_auc = roc_auc_score(y, oof_preds)
    print(f"LGBM OOF AUC: {oof_auc:.5f}")
    feature_importances_df['mean_importance'] =
feature_importances_df.mean(axis=1)
    feature_importances_df.sort_values(by='mean_importance',
ascending=False, inplace=True)
    print("\nLGBM Top 30 Feature Importances (Mean over folds):")
    print(feature_importances_df[['mean_importance']].head(30))
    return test_preds, oof_auc, feature_importances_df

# --- Deep Learning Model (PyTorch) ---
if PYTORCH_AVAILABLE:
    # This is a helper to organize the data (categorical features,
numerical features, and the target labels)
    #  in a way PyTorch can easily use, especially for feeding data in
batches during training.
    class TianchiDataset(Dataset):
        """Custom PyTorch Dataset for handling mixed data types."""
        def __init__(self, cat_features, num_features, labels=None):
            self.cat_features = {k: torch.tensor(v, dtype=torch.long)
for k, v in cat_features.items()}
            self.num_features = torch.tensor(num_features,
dtype=torch.float32)
            self.labels = torch.tensor(labels, dtype=torch.float32) if
labels is not None else None

        def __len__(self):
            # Assume all categorical features have the same length,
pick one
            return len(self.num_features)

        def __getitem__(self, idx):
```

```
            cat_item = {k: v[idx] for k, v in
self.cat_features.items()}
            num_item = self.num_features[idx]
            if self.labels is not None:
                return (cat_item, num_item),
self.labels[idx].unsqueeze(-1) # Ensure label is [batch_size, 1]
            else:
                return (cat_item, num_item)

    class DeepNet(nn.Module):
        # This class builds the actual neural network.
        """PyTorch Deep Learning Model for tabular data."""
        # It takes categorical features (like merchant_id, age_range)
and turns them into dense vector representations called "embeddings."
This helps the model understand relationships between different
categories.
        def __init__(self, embedding_info, num_numerical_features,
hidden_dims=[512, 256, 128], dropout_rates=[0.4, 0.4, 0.3]):
            super(DeepNet, self).__init__()
            self.embeddings = nn.ModuleList()
            total_embedding_dim = 0
            for col_name, input_dim, output_dim in embedding_info:
                self.embeddings.append(nn.Embedding(input_dim,
output_dim))
                total_embedding_dim += output_dim

            self.embedding_dropout = nn.Dropout(0.2) # Dropout after
embedding concatenation
            # It then combines these learned embeddings with the
regular numerical features.
            # Dense layers
            all_input_dims = total_embedding_dim +
num_numerical_features
            # This combined information is passed through several
"dense layers" (standard neural network layers) with techniques like
BatchNormalization (to stabilize learning) and Dropout (to prevent
overfitting).
            layers = []
            for i, hidden_dim in enumerate(hidden_dims):
                layers.append(nn.Linear(all_input_dims if i == 0 else
hidden_dims[i-1], hidden_dim))
                layers.append(nn.BatchNorm1d(hidden_dim))
                layers.append(nn.ReLU())
                layers.append(nn.Dropout(dropout_rates[i]))

            self.dense_layers = nn.Sequential(*layers)
            # The last layer outputs a single probability (between 0
and 1) that the user will be loyal.
            self.output_layer = nn.Linear(hidden_dims[-1] if
```

```
hidden_dims else all_input_dims, 1)

        def forward(self, x_cat, x_num):
            embedded_cats = []
            # x_cat is a dictionary: {'col_name': tensor_data, ...}
            # self.embeddings is a ModuleList, need to iterate
carefully or name them
            # Assuming embedding_info provides names in the same order
as self.embeddings
            for i, col_name in enumerate(x_cat.keys()): # Iterate
through input categorical feature names
                 embedded_cats.append(self.embeddings[i]
(x_cat[col_name]))

            if embedded_cats:
                embedded_cats_concat = torch.cat(embedded_cats, dim=1)
                embedded_cats_concat =
self.embedding_dropout(embedded_cats_concat)
                x = torch.cat([embedded_cats_concat, x_num], dim=1)
            else:
                x = x_num

            x = self.dense_layers(x)
            x = torch.sigmoid(self.output_layer(x))
            return x

    def train_predict_deep_model(train_df, test_df,
categorical_cols_embed, numerical_cols, target_col='label',
n_splits=5, epochs=50, batch_size=1024):
        if not PYTORCH_AVAILABLE:
            print("PyTorch not available. Skipping Deep Learning
model.")
            return np.zeros(len(test_df)), 0.0

        print("Preparing data for PyTorch Deep Learning model...")
        # Data preparation for PyTorch
        # Store encoders and embedding info globally for consistent
test set transformation
        label_encoders = {}
        embedding_info_list = []

        # Prepare categorical features for PyTorch
        X_cat_train_processed = {}
        X_cat_test_processed = {}

        for col in tqdm(categorical_cols_embed, desc="Label Encoding
DL Categoricals"):
            # Combine train and test for fitting encoder to see all
possible values
            combined_data = pd.concat([train_df[col], test_df[col]],
```

```
axis=0).astype(str).fillna('__MISSING__')
            encoder = LabelEncoder()
            encoder.fit(combined_data)
            label_encoders[col] = encoder # Store encoder

            X_cat_train_processed[col] =
encoder.transform(train_df[col].astype(str).fillna('__MISSING__'))
            X_cat_test_processed[col] =
encoder.transform(test_df[col].astype(str).fillna('__MISSING__'))

            input_dim = len(encoder.classes_) # Number of unique
categories
            output_dim = min(50, (input_dim + 1) // 2) # Heuristic for
embedding output dimension
            embedding_info_list.append((col, input_dim, output_dim))
            print(f"  DL Cat Feature: {col}, Input Dim: {input_dim},
Output Dim: {output_dim}")

        y_train_dl = train_df[target_col].values
        oof_preds_dl = np.zeros(len(train_df))
        test_preds_dl = np.zeros(len(test_df))

        device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
        print(f"Using device: {device}")
        # Stratified K-Fold for Deep Learning
        skf = StratifiedKFold(n_splits=n_splits, shuffle=True,
random_state=123)
        # Training loop
        for fold, (train_idx, val_idx) in
enumerate(tqdm(skf.split(train_df, y_train_dl), total=n_splits,
desc="DL Folds")):
            print(f"--- DL Fold {fold+1}/{n_splits} ---")

            # Numerical features scaling for this fold
            scaler = StandardScaler()
            X_num_train_fold_scaled =
scaler.fit_transform(train_df.iloc[train_idx]
[numerical_cols].astype(np.float32))
            X_num_val_fold_scaled =
scaler.transform(train_df.iloc[val_idx]
[numerical_cols].astype(np.float32))
            X_num_test_fold_scaled =
scaler.transform(test_df[numerical_cols].astype(np.float32)) # Scale
test set

            # Prepare Keras inputs for this fold
            fold_X_cat_train = {col: X_cat_train_processed[col]
[train_idx] for col in categorical_cols_embed}
            fold_X_cat_val = {col: X_cat_train_processed[col][val_idx]
```

```
for col in categorical_cols_embed}

            train_dataset = TianchiDataset(fold_X_cat_train,
X_num_train_fold_scaled, y_train_dl[train_idx])
            val_dataset = TianchiDataset(fold_X_cat_val,
X_num_val_fold_scaled, y_train_dl[val_idx])
            # Create DataLoader for batching
            train_loader = DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)
            val_loader = DataLoader(val_dataset, batch_size=batch_size
* 2, shuffle=False)
            # Initialize the model, optimizer, and loss function (Adam
optimizer and binary cross-entropy loss)
            model = DeepNet(embedding_info_list,
len(numerical_cols)).to(device)
            optimizer = optim.Adam(model.parameters(), lr=0.001)
            criterion = nn.BCELoss() # Binary Cross Entropy for binary
classification

            best_val_auc = -1
            patience_counter = 0
            patience_epochs = 10 # For early stopping
            # lower the learning rate if no improvement in validation
AUC
            # This is similar to ReduceLROnPlateau in Keras, but we
will use a custom scheduler
            # ReduceLROnPlateau equivalent
            scheduler =
optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
factor=0.2, patience=5, verbose=True, min_lr=1e-6)
            # Early stopping based on validation AUC
            for epoch in range(epochs):
                model.train()
                train_loss_epoch = 0
                for (cat_batch, num_batch), labels_batch in
train_loader:
                    cat_batch = {k: v.to(device) for k,v in
cat_batch.items()}
                    num_batch, labels_batch = num_batch.to(device),
labels_batch.to(device)

                    optimizer.zero_grad()
                    outputs = model(cat_batch, num_batch)
                    loss = criterion(outputs, labels_batch)
                    loss.backward()
                    optimizer.step()
                    train_loss_epoch += loss.item()

                model.eval()
```

```
                val_preds_epoch = []
                val_labels_epoch = []
                with torch.no_grad():
                    for (cat_batch, num_batch), labels_batch in
val_loader:
                        cat_batch = {k: v.to(device) for k,v in
cat_batch.items()}
                        num_batch = num_batch.to(device)
                        outputs = model(cat_batch, num_batch)

val_preds_epoch.extend(outputs.cpu().numpy().ravel())

val_labels_epoch.extend(labels_batch.cpu().numpy().ravel())

                current_val_auc = roc_auc_score(val_labels_epoch,
val_preds_epoch)
                scheduler.step(current_val_auc) # For
ReduceLROnPlateau

                print(f"Epoch {epoch+1}/{epochs} - Train Loss:
{train_loss_epoch/len(train_loader):.4f} - Val AUC:
{current_val_auc:.4f} - LR: {optimizer.param_groups[0]['lr']:.1e}")

                if current_val_auc > best_val_auc:
                    best_val_auc = current_val_auc
                    torch.save(model.state_dict(),
DL_MODEL_CHECKPOINT_PATH.format(fold=fold+1))
                    print(f"  Best val_auc improved to
{best_val_auc:.4f}, model saved.")
                    patience_counter = 0
                else:
                    patience_counter += 1

                if patience_counter >= patience_epochs:
                    print("  Early stopping triggered.")
                    break

            # Load best model for OOF and test predictions

model.load_state_dict(torch.load(DL_MODEL_CHECKPOINT_PATH.format(fold=
fold+1)))
            model.eval()
            # Out-of-Fold predictions for this fold
            val_preds_list = []
            with torch.no_grad():
                for (cat_batch, num_batch), _ in val_loader: # Use
val_loader again for consistency
                    cat_batch = {k: v.to(device) for k,v in
cat_batch.items()}
                    num_batch = num_batch.to(device)
```

```
                    outputs = model(cat_batch, num_batch)

val_preds_list.extend(outputs.cpu().numpy().ravel())
            oof_preds_dl[val_idx] = val_preds_list[:len(val_idx)] #
Ensure correct length

            # Test predictions for this fold
            # Do the averaging across folds
            fold_test_cat_inputs = {col: X_cat_test_processed[col] for
col in categorical_cols_embed}
            test_dataset_fold = TianchiDataset(fold_test_cat_inputs,
X_num_test_fold_scaled) # No labels for test
            test_loader_fold = DataLoader(test_dataset_fold,
batch_size=batch_size*2, shuffle=False)

            current_fold_test_preds = []
            with torch.no_grad():
                for (cat_batch, num_batch) in test_loader_fold:
                    cat_batch = {k: v.to(device) for k,v in
cat_batch.items()}
                    num_batch = num_batch.to(device)
                    outputs = model(cat_batch, num_batch)

current_fold_test_preds.extend(outputs.cpu().numpy().ravel())
            test_preds_dl += np.array(current_fold_test_preds) /
n_splits

            del model, train_loader, val_loader, test_loader_fold,
X_num_train_fold_scaled, X_num_val_fold_scaled, X_num_test_fold_scaled
            gc.collect()
            if torch.cuda.is_available(): torch.cuda.empty_cache()
        # Final output predictions and overall OOF AUC
        overall_oof_auc_dl = roc_auc_score(y_train_dl, oof_preds_dl)
        print(f"Overall DL OOF AUC: {overall_oof_auc_dl:.5f}")
        return test_preds_dl, overall_oof_auc_dl
else: # PYTORCH_AVAILABLE is False
    def train_predict_deep_model(*args, **kwargs): # Stub if PyTorch
not available
        print("PyTorch is not installed. Skipping Deep Learning model
training.")
        test_df_len = kwargs.get('test_df', pd.DataFrame()).shape[0] #
Get length of test_df if passed
        if 'train_df' in kwargs: test_df_len =
kwargs['train_df'].shape[0] # Fallback for OOF shape
        return np.zeros(test_df_len), 0.0


# --- Main Execution ---
if __name__ == '__main__':
    # start, and time the script
```

```
    script_start_time = datetime.now()
    print(f"Competition script started at: {script_start_time}")
    # Load data
    user_log_df, user_info_df, train_target_df, test_target_df =
load_data()
    # Create Features with the function above
    train_featured_df, test_featured_df = create_all_features(
        user_log_df, user_info_df, train_target_df, test_target_df
    )
    del user_log_df, user_info_df; gc.collect()

    print(f"Train featured shape: {train_featured_df.shape}")
    print(f"Test featured shape: {test_featured_df.shape}")
    # Define the target column and drop unnecessary columns for model
definition
    label_col = 'label'
    cols_to_drop_for_model_definition = [label_col, 'prob', 'user_id']

    all_engineered_cols = [col for col in train_featured_df.columns if
col not in cols_to_drop_for_model_definition]

    # Define categorical and numerical features for DL
    # These lists should be carefully curated based on feature
understanding
    potential_cat_cols_for_dl = ['u_age_range', 'u_gender',
'merchant_id',
                                 'acq_item_id', 'acq_cat_id',
'acq_brand_id']
    # Add more if they are truly categorical and suitable for
embeddings
    # e.g., if action type counts are binned or treated as categories.
    # Separate categorical and numerical features
    categorical_features_for_dl_embed = [col for col in
potential_cat_cols_for_dl if col in all_engineered_cols]
    numerical_features_for_dl = [col for col in all_engineered_cols if
col not in categorical_features_for_dl_embed]

    print(f"Identified {len(categorical_features_for_dl_embed)}
categorical features for DL embeddings:
{categorical_features_for_dl_embed}")
    print(f"Identified {len(numerical_features_for_dl)} numerical
features for DL: {numerical_features_for_dl[:10]}...")
    # Train LightGBM and do predictions
    # --- LightGBM Model ---
    lgbm_features_to_use = all_engineered_cols
    print(f"Using {len(lgbm_features_to_use)} features for LGBM
training.")

    lgbm_test_preds, lgbm_oof_auc, _ = train_predict_lgbm(
        train_featured_df.copy(),
```

```
        test_featured_df.copy(),
        lgbm_features_to_use,
        target_col=label_col
    )
    # Train PyTorch Deep Learning model and do predictions
    # --- Deep Learning Model (PyTorch) ---
    dl_test_preds = None
    dl_oof_auc = 0.0 # Default if DL is skipped
    if PYTORCH_AVAILABLE:
        if not train_featured_df.empty and not test_featured_df.empty
and \
           (len(numerical_features_for_dl) > 0 or
len(categorical_features_for_dl_embed) > 0) : # Ensure there are
features

            dl_test_preds, dl_oof_auc = train_predict_deep_model(
                train_featured_df,
                test_featured_df,
                categorical_features_for_dl_embed,
                numerical_features_for_dl,
                target_col=label_col,
                n_splits=5,
                epochs=30, # Adjust epochs based on observed
convergence
                batch_size=2048 # Adjust batch size based on memory
and dataset size
            )
        else:
            print("Skipping DL model due to no features or empty
dataframes.")
    # Ensemble two models and do predictions by weighted average based
on two OOF AUC
    # --- Ensemble Predictions ---
    if dl_test_preds is not None and lgbm_test_preds is not None:
        print("Ensembling LGBM and PyTorch DL predictions...")
        # Simple average or weighted average based on OOF scores
        # Example: Weighted average, tune weights based on OOF scores
        total_oof_auc = lgbm_oof_auc + dl_oof_auc
        if total_oof_auc > 0:
            lgbm_weight = lgbm_oof_auc / total_oof_auc
            dl_weight = dl_oof_auc / total_oof_auc
        else: # Fallback if OOF AUCs are zero (e.g., if models failed
or data is problematic)
            lgbm_weight = 0.5
            dl_weight = 0.5

        print(f"LGBM OOF: {lgbm_oof_auc:.4f}, DL OOF:
{dl_oof_auc:.4f}")
        print(f"Ensemble Weights -> LGBM: {lgbm_weight:.3f}, DL:
```

```
{dl_weight:.3f}")
        final_preds = (lgbm_weight * lgbm_test_preds) + (dl_weight *
dl_test_preds)
    elif lgbm_test_preds is not None:
        print("Using only LGBM predictions.")
        final_preds = lgbm_test_preds
    else:
        print("No model predictions available. Generating dummy
submission (all zeros).")
        final_preds = np.zeros(len(test_target_df))
    # Save the final predictions to a CSV file
    # --- Create Submission File ---
    submission_df = test_target_df[['user_id', 'merchant_id']].copy()
    submission_df['prob'] = final_preds
    submission_df['prob'] = np.clip(submission_df['prob'], 0.0, 1.0)
    submission_df.to_csv(SUBMISSION_FILE, index=False, header=True)
    print(f"Submission file '{SUBMISSION_FILE}' created with
{len(submission_df)} rows.")
    print(f"Sample predictions:\n{submission_df.head()}")

    script_end_time = datetime.now()
    print(f"Script finished at: {script_end_time}. Total runtime:
{script_end_time - script_start_time}")
```

```
Competition script started at: 2025-05-11 22:48:42.836280
Loading data...
Preprocessing basic data...
Data loaded and basic preprocessing done.
Starting comprehensive feature engineering...
Engineering user-level features...
Engineering merchant-level features...
Engineering user-merchant interaction features...

C:\Users\yishu\AppData\Local\Temp\ipykernel_61768\1505742554.py:130:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  else: merged_df[col].fillna(0, inplace=True)
C:\Users\yishu\AppData\Local\Temp\ipykernel_61768\1505742554.py:128:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
```

```
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  if 'days_prior' in col or 'score' in col: merged_df[col].fillna(-1,
inplace=True)
C:\Users\yishu\AppData\Local\Temp\ipykernel_61768\1505742554.py:129:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  elif col in ['acq_item_id', 'acq_cat_id', 'acq_brand_id']:
merged_df[col].fillna(0, inplace=True)

Feature engineering complete.
Train featured shape: (260864, 89)
Test featured shape: (261477, 89)
Identified 6 categorical features for DL embeddings: ['u_age_range',
'u_gender', 'merchant_id', 'acq_item_id', 'acq_cat_id',
'acq_brand_id']
Identified 81 numerical features for DL: ['u_hist_total_actions',
'u_hist_n_distinct_items', 'u_hist_n_distinct_categories',
'u_hist_n_distinct_merchants', 'u_hist_n_distinct_brands',
'u_hist_days_active', 'u_hist_earliest_action_days_prior',
'u_hist_latest_action_days_prior', 'u_hist_mean_action_days_prior',
'u_hist_std_action_days_prior']...
Using 87 features for LGBM training.
Training LightGBM model...

LGBM Folds: 100%|██████████| 5/5 [00:29<00:00,  5.84s/it]

LGBM OOF AUC: 0.68686

LGBM Top 30 Feature Importances (Mean over folds):
                                            mean_importance
acq_item_id                                          1337.2
merchant_id                                          1023.4
```

```
acq_brand_id                                                927.6
acq_cat_id                                                  635.2
ratio_um_d11_actions_vs_m_d11_interactions                  598.0
ratio_um_d11_actions_vs_u_d11_actions                       313.4
um_hist_first_interaction_days_prior                        287.8
u_hist_purchase_to_click_ratio                              273.0
u_hist_mean_action_days_prior                               258.6
u_hist_std_action_days_prior                                246.6
m_d11_action_type_2_count                                   230.8
u_hist_purchase_to_fav_ratio                                216.4
um_d11_distinct_items_interacted                            213.8
um_d11_purchases                                            210.8
u_hist_earliest_action_days_prior                           183.6
u_hist_purchase_to_cart_ratio                               182.6
ratio_um_hist_actions_vs_u_hist_actions                     180.6
u_hist_days_active                                          163.2
u_d11_n_distinct_merchants                                  161.6
u_purchases_last_180d_prior                                 160.2
m_d11_total_interactions                                    155.2
um_hist_distinct_items                                      153.2
u_d11_total_actions                                         148.4
m_d11_n_distinct_users                                      146.6
u_hist_action_type_2_count                                  142.4
ratio_um_d11_purchases_vs_u_d11_purchases                   142.2
u_d11_action_type_0_count                                   141.4
u_actions_last_90d_prior                                    136.8
um_d11_total_actions                                        130.4
u_hist_action_type_3_count                                  127.4
Preparing data for PyTorch Deep Learning model...

Label Encoding DL Categoricals:  17%|█▋        | 1/6 [00:00<00:00,
6.21it/s]

  DL Cat Feature: u_age_range, Input Dim: 9, Output Dim: 5

Label Encoding DL Categoricals:  33%|███▎      | 2/6 [00:00<00:00,
6.39it/s]

  DL Cat Feature: u_gender, Input Dim: 3, Output Dim: 2

Label Encoding DL Categoricals:  50%|█████     | 3/6 [00:00<00:00,
5.86it/s]

  DL Cat Feature: merchant_id, Input Dim: 1994, Output Dim: 50

Label Encoding DL Categoricals:  83%|████████▎ | 5/6 [00:00<00:00,
4.73it/s]

  DL Cat Feature: acq_item_id, Input Dim: 71478, Output Dim: 50
  DL Cat Feature: acq_cat_id, Input Dim: 966, Output Dim: 50
```

```
Label Encoding DL Categoricals: 100%|██████████| 6/6 [00:01<00:00,
5.04it/s]

  DL Cat Feature: acq_brand_id, Input Dim: 2865, Output Dim: 50
Using device: cuda

DL Folds:   0%|          | 0/5 [00:00<?, ?it/s]

--- DL Fold 1/5 ---

c:\Users\yishu\AppData\Local\Programs\Python\Python313\Lib\site-
packages\torch\optim\lr_scheduler.py:62: UserWarning: The verbose
parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn(

Epoch 1/30 - Train Loss: 0.2880 - Val AUC: 0.6588 - LR: 1.0e-03
  Best val_auc improved to 0.6588, model saved.
Epoch 2/30 - Train Loss: 0.2266 - Val AUC: 0.6677 - LR: 1.0e-03
  Best val_auc improved to 0.6677, model saved.
Epoch 3/30 - Train Loss: 0.2230 - Val AUC: 0.6731 - LR: 1.0e-03
  Best val_auc improved to 0.6731, model saved.
Epoch 4/30 - Train Loss: 0.2208 - Val AUC: 0.6798 - LR: 1.0e-03
  Best val_auc improved to 0.6798, model saved.
Epoch 5/30 - Train Loss: 0.2194 - Val AUC: 0.6824 - LR: 1.0e-03
  Best val_auc improved to 0.6824, model saved.
Epoch 6/30 - Train Loss: 0.2176 - Val AUC: 0.6780 - LR: 1.0e-03
Epoch 7/30 - Train Loss: 0.2164 - Val AUC: 0.6790 - LR: 1.0e-03
Epoch 8/30 - Train Loss: 0.2146 - Val AUC: 0.6824 - LR: 1.0e-03
  Best val_auc improved to 0.6824, model saved.
Epoch 9/30 - Train Loss: 0.2125 - Val AUC: 0.6799 - LR: 1.0e-03
Epoch 10/30 - Train Loss: 0.2109 - Val AUC: 0.6792 - LR: 1.0e-03
Epoch 11/30 - Train Loss: 0.2095 - Val AUC: 0.6747 - LR: 2.0e-04
Epoch 12/30 - Train Loss: 0.2055 - Val AUC: 0.6757 - LR: 2.0e-04
Epoch 13/30 - Train Loss: 0.2045 - Val AUC: 0.6749 - LR: 2.0e-04
Epoch 14/30 - Train Loss: 0.2046 - Val AUC: 0.6752 - LR: 2.0e-04
Epoch 15/30 - Train Loss: 0.2037 - Val AUC: 0.6745 - LR: 2.0e-04
Epoch 16/30 - Train Loss: 0.2032 - Val AUC: 0.6737 - LR: 2.0e-04
Epoch 17/30 - Train Loss: 0.2021 - Val AUC: 0.6731 - LR: 4.0e-05
Epoch 18/30 - Train Loss: 0.2013 - Val AUC: 0.6731 - LR: 4.0e-05
  Early stopping triggered.

DL Folds:  20%|██        | 1/5 [02:12<08:50, 132.71s/it]

--- DL Fold 2/5 ---

c:\Users\yishu\AppData\Local\Programs\Python\Python313\Lib\site-
packages\torch\optim\lr_scheduler.py:62: UserWarning: The verbose
parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn(
```

```
Epoch 1/30 - Train Loss: 0.2958 - Val AUC: 0.6529 - LR: 1.0e-03
  Best val_auc improved to 0.6529, model saved.
Epoch 2/30 - Train Loss: 0.2266 - Val AUC: 0.6664 - LR: 1.0e-03
  Best val_auc improved to 0.6664, model saved.
Epoch 3/30 - Train Loss: 0.2232 - Val AUC: 0.6660 - LR: 1.0e-03
Epoch 4/30 - Train Loss: 0.2211 - Val AUC: 0.6774 - LR: 1.0e-03
  Best val_auc improved to 0.6774, model saved.
Epoch 5/30 - Train Loss: 0.2193 - Val AUC: 0.6817 - LR: 1.0e-03
  Best val_auc improved to 0.6817, model saved.
Epoch 6/30 - Train Loss: 0.2172 - Val AUC: 0.6827 - LR: 1.0e-03
  Best val_auc improved to 0.6827, model saved.
Epoch 7/30 - Train Loss: 0.2158 - Val AUC: 0.6826 - LR: 1.0e-03
Epoch 8/30 - Train Loss: 0.2142 - Val AUC: 0.6827 - LR: 1.0e-03
  Best val_auc improved to 0.6827, model saved.
Epoch 9/30 - Train Loss: 0.2113 - Val AUC: 0.6807 - LR: 1.0e-03
Epoch 10/30 - Train Loss: 0.2095 - Val AUC: 0.6795 - LR: 1.0e-03
Epoch 11/30 - Train Loss: 0.2079 - Val AUC: 0.6784 - LR: 1.0e-03
Epoch 12/30 - Train Loss: 0.2053 - Val AUC: 0.6748 - LR: 2.0e-04
Epoch 13/30 - Train Loss: 0.2019 - Val AUC: 0.6748 - LR: 2.0e-04
Epoch 14/30 - Train Loss: 0.2008 - Val AUC: 0.6738 - LR: 2.0e-04
Epoch 15/30 - Train Loss: 0.2001 - Val AUC: 0.6722 - LR: 2.0e-04
Epoch 16/30 - Train Loss: 0.1992 - Val AUC: 0.6716 - LR: 2.0e-04
Epoch 17/30 - Train Loss: 0.1993 - Val AUC: 0.6695 - LR: 2.0e-04
Epoch 18/30 - Train Loss: 0.1983 - Val AUC: 0.6693 - LR: 4.0e-05
  Early stopping triggered.

DL Folds:  40%|████      | 2/5 [04:30<06:47, 135.67s/it]

--- DL Fold 3/5 ---

c:\Users\yishu\AppData\Local\Programs\Python\Python313\Lib\site-
packages\torch\optim\lr_scheduler.py:62: UserWarning: The verbose
parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn(

Epoch 1/30 - Train Loss: 0.3200 - Val AUC: 0.6630 - LR: 1.0e-03
  Best val_auc improved to 0.6630, model saved.
Epoch 2/30 - Train Loss: 0.2270 - Val AUC: 0.6723 - LR: 1.0e-03
  Best val_auc improved to 0.6723, model saved.
Epoch 3/30 - Train Loss: 0.2241 - Val AUC: 0.6776 - LR: 1.0e-03
  Best val_auc improved to 0.6776, model saved.
Epoch 4/30 - Train Loss: 0.2220 - Val AUC: 0.6819 - LR: 1.0e-03
  Best val_auc improved to 0.6819, model saved.
Epoch 5/30 - Train Loss: 0.2203 - Val AUC: 0.6832 - LR: 1.0e-03
  Best val_auc improved to 0.6832, model saved.
Epoch 6/30 - Train Loss: 0.2179 - Val AUC: 0.6873 - LR: 1.0e-03
  Best val_auc improved to 0.6873, model saved.
Epoch 7/30 - Train Loss: 0.2167 - Val AUC: 0.6870 - LR: 1.0e-03
Epoch 8/30 - Train Loss: 0.2148 - Val AUC: 0.6882 - LR: 1.0e-03
```

```
  Best val_auc improved to 0.6882, model saved.
Epoch 9/30 - Train Loss: 0.2135 - Val AUC: 0.6883 - LR: 1.0e-03
  Best val_auc improved to 0.6883, model saved.
Epoch 10/30 - Train Loss: 0.2116 - Val AUC: 0.6869 - LR: 1.0e-03
Epoch 11/30 - Train Loss: 0.2098 - Val AUC: 0.6873 - LR: 1.0e-03
Epoch 12/30 - Train Loss: 0.2080 - Val AUC: 0.6863 - LR: 1.0e-03
Epoch 13/30 - Train Loss: 0.2057 - Val AUC: 0.6858 - LR: 1.0e-03
Epoch 14/30 - Train Loss: 0.2039 - Val AUC: 0.6850 - LR: 1.0e-03
Epoch 15/30 - Train Loss: 0.2014 - Val AUC: 0.6777 - LR: 2.0e-04
Epoch 16/30 - Train Loss: 0.1975 - Val AUC: 0.6789 - LR: 2.0e-04
Epoch 17/30 - Train Loss: 0.1964 - Val AUC: 0.6785 - LR: 2.0e-04
Epoch 18/30 - Train Loss: 0.1953 - Val AUC: 0.6770 - LR: 2.0e-04
Epoch 19/30 - Train Loss: 0.1950 - Val AUC: 0.6767 - LR: 2.0e-04
  Early stopping triggered.

DL Folds:  60%|██████    | 3/5 [06:52<04:37, 138.67s/it]

--- DL Fold 4/5 ---

c:\Users\yishu\AppData\Local\Programs\Python\Python313\Lib\site-
packages\torch\optim\lr_scheduler.py:62: UserWarning: The verbose
parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn(

Epoch 1/30 - Train Loss: 0.3040 - Val AUC: 0.6649 - LR: 1.0e-03
  Best val_auc improved to 0.6649, model saved.
Epoch 2/30 - Train Loss: 0.2264 - Val AUC: 0.6719 - LR: 1.0e-03
  Best val_auc improved to 0.6719, model saved.
Epoch 3/30 - Train Loss: 0.2236 - Val AUC: 0.6740 - LR: 1.0e-03
  Best val_auc improved to 0.6740, model saved.
Epoch 4/30 - Train Loss: 0.2218 - Val AUC: 0.6808 - LR: 1.0e-03
  Best val_auc improved to 0.6808, model saved.
Epoch 5/30 - Train Loss: 0.2199 - Val AUC: 0.6832 - LR: 1.0e-03
  Best val_auc improved to 0.6832, model saved.
Epoch 6/30 - Train Loss: 0.2180 - Val AUC: 0.6866 - LR: 1.0e-03
  Best val_auc improved to 0.6866, model saved.
Epoch 7/30 - Train Loss: 0.2169 - Val AUC: 0.6887 - LR: 1.0e-03
  Best val_auc improved to 0.6887, model saved.
Epoch 8/30 - Train Loss: 0.2152 - Val AUC: 0.6860 - LR: 1.0e-03
Epoch 9/30 - Train Loss: 0.2132 - Val AUC: 0.6860 - LR: 1.0e-03
Epoch 10/30 - Train Loss: 0.2114 - Val AUC: 0.6863 - LR: 1.0e-03
Epoch 11/30 - Train Loss: 0.2099 - Val AUC: 0.6819 - LR: 1.0e-03
Epoch 12/30 - Train Loss: 0.2079 - Val AUC: 0.6820 - LR: 1.0e-03
Epoch 13/30 - Train Loss: 0.2059 - Val AUC: 0.6782 - LR: 2.0e-04
Epoch 14/30 - Train Loss: 0.2014 - Val AUC: 0.6792 - LR: 2.0e-04
Epoch 15/30 - Train Loss: 0.2011 - Val AUC: 0.6769 - LR: 2.0e-04
Epoch 16/30 - Train Loss: 0.1999 - Val AUC: 0.6768 - LR: 2.0e-04
Epoch 17/30 - Train Loss: 0.1995 - Val AUC: 0.6762 - LR: 2.0e-04
  Early stopping triggered.
```

```
DL Folds:  80%|████████  | 4/5 [08:54<02:12, 132.09s/it]

--- DL Fold 5/5 ---

c:\Users\yishu\AppData\Local\Programs\Python\Python313\Lib\site-
packages\torch\optim\lr_scheduler.py:62: UserWarning: The verbose
parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn(

Epoch 1/30 - Train Loss: 0.3313 - Val AUC: 0.6494 - LR: 1.0e-03
  Best val_auc improved to 0.6494, model saved.
Epoch 2/30 - Train Loss: 0.2272 - Val AUC: 0.6646 - LR: 1.0e-03
  Best val_auc improved to 0.6646, model saved.
Epoch 3/30 - Train Loss: 0.2240 - Val AUC: 0.6722 - LR: 1.0e-03
  Best val_auc improved to 0.6722, model saved.
Epoch 4/30 - Train Loss: 0.2213 - Val AUC: 0.6754 - LR: 1.0e-03
  Best val_auc improved to 0.6754, model saved.
Epoch 5/30 - Train Loss: 0.2199 - Val AUC: 0.6778 - LR: 1.0e-03
  Best val_auc improved to 0.6778, model saved.
Epoch 6/30 - Train Loss: 0.2184 - Val AUC: 0.6763 - LR: 1.0e-03
Epoch 7/30 - Train Loss: 0.2162 - Val AUC: 0.6776 - LR: 1.0e-03
Epoch 8/30 - Train Loss: 0.2150 - Val AUC: 0.6753 - LR: 1.0e-03
Epoch 9/30 - Train Loss: 0.2130 - Val AUC: 0.6753 - LR: 1.0e-03
Epoch 10/30 - Train Loss: 0.2109 - Val AUC: 0.6769 - LR: 1.0e-03
Epoch 11/30 - Train Loss: 0.2099 - Val AUC: 0.6750 - LR: 2.0e-04
Epoch 12/30 - Train Loss: 0.2061 - Val AUC: 0.6743 - LR: 2.0e-04
Epoch 13/30 - Train Loss: 0.2054 - Val AUC: 0.6739 - LR: 2.0e-04
Epoch 14/30 - Train Loss: 0.2039 - Val AUC: 0.6729 - LR: 2.0e-04
Epoch 15/30 - Train Loss: 0.2037 - Val AUC: 0.6718 - LR: 2.0e-04
  Early stopping triggered.

DL Folds: 100%|██████████| 5/5 [10:43<00:00, 128.63s/it]

Overall DL OOF AUC: 0.68362
Ensembling LGBM and PyTorch DL predictions...
LGBM OOF: 0.6869, DL OOF: 0.6836
Ensemble Weights -> LGBM: 0.501, DL: 0.499
Submission file 'prediction_pytorch_lgbm.csv' created with 261477
rows.
Sample predictions:
   user_id  merchant_id      prob
0   163968         4605  0.073557
1   360576         1581  0.115936
2    98688         1964  0.077613
3    98688         3645  0.069432
4   295296         3361  0.084315
Script finished at: 2025-05-11 23:08:53.383661. Total runtime:
0:20:10.547381
```