VIDEO 3 – DATA PREPROCESSING

DEEP LINE-BY-LINE CODE EXPLANATION

' ==
' BLOCK 1: IMPORTS, PROJECT CONFIGURATION & RANDOM SEED
' Purpose:
' - Prepare the Python environment
' - Define project-level settings
' - Fix randomness for reproducibility
' ==

' ---------- Import Required Libraries ----------

◇ import os
' Used to interact with the operating system.
' Helps control system-level settings like randomness.

◇ import gc
' Used for garbage collection.
' Frees unused memory, important for large datasets.

◇ import random
' Used to generate random numbers.
' Machine learning uses randomness internally.

◇ import warnings
' Used to control warning messages.

◇ import numpy as np
' Used for numerical operations and array handling.

◇ import pandas as pd
' Main library for working with tabular data (rows and columns).
' Used extensively in data preprocessing.

◇ import lightgbm as lgb
' Machine learning library.
' Imported here for later model training.

' ---------- Suppress Warning Messages ----------

◇ warnings.filterwarnings("ignore")
' Hides warning messages to keep output clean and readable.

' ---------- Project Configuration Class ----------

◇ class ProjectConfig:
' Stores all important project settings in one place.

◇ DATA_PATH = "D:/M5 Data"
' Folder location where all dataset CSV files are stored.

◇ TRAIN_END = 1913
' Last day of known sales data.
' Training uses data up to day 1913.

◇ FORECAST_HORIZON = 28
' Number of future days to predict.

◇ RANDOM_STATE = 42
' Fixed value to control randomness.
' Ensures same results every time the code runs.

◇ LGB_PARAMS = { ... }
' Dictionary containing LightGBM model parameters.
' Used later during model training.

' ---------- Fixing Randomness ----------

◇ def seed_everything(seed = 42):
' Function to fix randomness across the entire project.

◇ random.seed(seed)
' Fixes randomness for Python.

◇ np.random.seed(seed)
' Fixes randomness for NumPy operations.

◇ os.environ["PYTHONHASHSEED"] = str(seed)
' Fixes internal Python hashing behavior.

◇ seed_everything(ProjectConfig.RANDOM_STATE)
' Calls the function to apply randomness control.

' ---------- Block 1 Summary ----------
' This block prepares the environment by:
' - Importing required libraries
' - Defining project configuration
' - Fixing randomness for reproducibility
' ==

' ==
' BLOCK 2: LOADING DATASETS
' Purpose:
' - Load raw CSV files into memory
' - Convert them into table format (DataFrames)
' - Verify dataset sizes
' ==

' ---------- Load Sales Dataset ----------

◇ sales = pd.read_csv(f"{ProjectConfig.DATA_PATH}/sales_train_validation.csv")
' Reads the sales CSV file from the dataset folder.
' Contains daily sales data for each product and store.
' Data is loaded into a pandas DataFrame.

' ---------- Load Calendar Dataset ----------

◇ calendar = pd.read_csv(f"{ProjectConfig.DATA_PATH}/calendar.csv")
' Reads the calendar CSV file.
' Contains date information, holidays, and events.
' Used to add time-based context to sales data.

' ---------- Load Prices Dataset ----------

◇ prices = pd.read_csv(f"{ProjectConfig.DATA_PATH}/sell_prices.csv")
' Reads the prices CSV file.
' Contains weekly selling prices for each product and store.
' Used to understand price-demand relationships.

' ---------- Verify Dataset Sizes ----------

◇ print("Sales shape:", sales.shape)
' Displays number of rows and columns in sales dataset.
' Helps understand the size of sales data.

◇ print("Calendar shape:", calendar.shape)
' Displays size of calendar dataset.

◇ print("Prices shape:", prices.shape)
' Displays size of prices dataset.

' ---------- Block 2 Summary ----------
' This block loads all raw datasets into memory
' and verifies their dimensions before preprocessing.
' ==

' ===
' BLOCK 3: MEMORY OPTIMIZATION USING DOWNCASTING
' Purpose:
' - Reduce RAM usage of large datasets
' - Prevent system crashes
' - Make processing faster and efficient
' ===

◇ def downcast_dtypes(df, verbose = True):
' This function reduces memory usage of a DataFrame.
' It converts large data types into smaller ones
' without losing information.

◇ start_mem = df.memory_usage().sum() / 1024**2
' Calculates total memory used by the dataset (in MB)
' before optimization.

◇ numerics = ['int16','int32','int64','float16','float32','float64']
' List of numeric data types we want to optimize.
' Only numeric columns are considered.

◇ for col in df.columns:
' Loops through each column in the dataset one by one.

◇ col_type = df[col].dtypes
' Stores the data type of the current column.

◇ if col_type in numerics:
' Checks whether the column is numeric.
' Non-numeric columns are skipped.

◇ c_min = df[col].min()
◇ c_max = df[col].max()
' Finds minimum and maximum values in the column.
' Used to decide the smallest safe data type.

◇ if str(col_type).startswith("int"):
' Checks whether the column contains integer values.

◇ df[col] = df[col].astype(np.int8)
' Converts column to int8 if values fit within range.

' Uses the smallest possible integer type.

◇ elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
' If values do not fit in int8, try int16.

◇ df[col] = df[col].astype(np.int16)
' Converts column to int16 safely.

◇ elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
' If needed, convert to int32.

◇ df[col] = df[col].astype(np.int32)
' Uses int32 for larger integer ranges.

◇ else:
' If values are very large, keep int64.

◇ df[col] = df[col].astype(np.int64)
' Keeps original integer type when required.

◇ else:
' This block handles floating-point numbers.

◇ df[col] = df[col].astype(np.float16)
' Converts float values to float16 if safe.
' Saves significant memory.

◇ elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).max:
' If float16 is not sufficient, try float32.

◇ df[col] = df[col].astype(np.float32)
' Converts to float32 safely.

◇ else:
' If values are too large, keep float64.

◇ df[col] = df[col].astype(np.float64)
' Retains original floating-point precision.

◇ end_mem = df.memory_usage().sum() / 1024**2

' Calculates memory usage after optimization.

◇ if verbose:

' Checks whether memory usage details should be printed.

◇ print(f"Memory usage dropped to {end_mem:.2f} MB")

' Displays how much memory is being used now.
' Helps confirm optimization effectiveness.

◇ return df

' Returns the optimized DataFrame.

' ---------- Block 3 Summary ----------
' This block reduces memory usage by:
' - Identifying numeric columns
' - Converting them to smaller data types
' - Making large datasets manageable
' ==

' ==
' BLOCK 4: READ DATA WITH MEMORY OPTIMIZATION
' Purpose:
' - Load all raw datasets from disk
' - Immediately optimize memory usage
' - Return clean, efficient DataFrames
' ==

◇ def read_data(path):

' This function loads all required CSV files
' from the given folder path.

◇ print(f"Reading files from {path}...")

' Displays a message showing from where
' the data is being read.
' Useful for tracking execution progress.

' ---------- Load and Optimize Calendar Data ----------

◆ calendar = pd.read_csv(f"{path}/calendar.csv")

' Reads the calendar CSV file.

' Contains date, event, and holiday information.

◇ calendar = downcast_dtypes(calendar)
' Applies memory optimization to calendar data.
' Reduces RAM usage.

' ---------- Load and Optimize Price Data ----------

◇ prices = pd.read_csv(f"{path}/sell_prices.csv")
' Reads the sell prices CSV file.
' Contains weekly price information.

◇ prices = downcast_dtypes(prices)
' Optimizes memory usage for price data.

' ---------- Load and Optimize Sales Data ----------

◇ sales = pd.read_csv(f"{path}/sales_train_validation.csv")
' Reads the main sales dataset.
' Contains daily sales values for each product and store.

◇ sales = downcast_dtypes(sales)
' Optimizes memory usage for sales data.

' ---------- Return All Datasets ----------

◇ return sales, calendar, prices
' Returns all three datasets together.
' These datasets are now memory-efficient
' and ready for further preprocessing.

' ---------- Block 4 Summary ----------
' This block ensures that:
' - All datasets are loaded correctly
' - Memory optimization is applied early
' - Large retail data can be processed smoothly
' ==

' ==
' BLOCK 5: CALLING THE DATA LOADING FUNCTION
' Purpose:
' - Execute the read_data function
' - Load all datasets into memory
' - Store them in separate variables
' ==

◇ df_sales, df_calendar, df_prices = read_data(ProjectConfig.DATA_PATH)
' Calls the read_data function using the dataset folder path.

' This line performs three actions at once:
' 1. Reads the sales dataset
' 2. Reads the calendar dataset
' 3. Reads the prices dataset

' All three datasets are:
' - Loaded from disk
' - Memory-optimized using downcasting

' The results are stored in:
' df_sales → sales data
' df_calendar → calendar data
' df_prices → price data

' ---------- Block 5 Summary ----------
' This block executes the data loading pipeline
' and prepares all datasets for further preprocessing.
' ==

' ==
' BLOCK 6: TRANSFORM AND MERGE DATASETS
' Purpose:
' - Prepare data for machine learning
' - Convert sales data from wide to long format
' - Merge sales, calendar, and price information
' ==

◇ def transform_and_merge(sales, calendar, prices, config):
' This function prepares the final base dataset
' by reshaping and combining all raw datasets.

' ---------- Add Future Forecast Columns ----------

◇ for day in range(config.FORECAST_HORIZON):
' Loops over the forecast horizon (28 days).

◇ sales[f"d_{config.TRAIN_END + day + 1}"] = np.nan
' Adds future day columns (d_1914 to d_1941).
' These columns represent future dates with unknown sales.

' ---------- Select Recent Sales Columns ----------

◇ start_idx = max(1, config.TRAIN_END - 1000)
' Limits the data to recent days only.

' Reduces dataset size and speeds up processing.

◇ value_cols = [c for c in sales.columns if c.startswith("d_") and int(c.split("_")[1]) >= start_idx]
' Selects only sales columns (d_x) starting from start_idx.
' Helps focus on recent sales history.

' ---------- Identify Product Information Columns ----------

◇ id_cols = ["id", "item_id", "dept_id", "cat_id", "store_id", "state_id"]
' These columns identify the product, store, and state.
' They are kept unchanged during transformation.

' ---------- Convert Wide Data to Long Format ----------

◇ data = pd.melt(sales, id_vars=id_cols, value_vars=value_cols, var_name="d", value_name="sales")
' Converts sales data from wide format to long format.

' After this step:
' - Each row represents one product on one day
' - This format is required for time-series modeling

' ---------- Merge Calendar Data ----------

◇ calendar = calendar.drop(["weekday", "wday", "month", "year"], axis=1)
' Removes unnecessary calendar columns.
' Reduces memory usage and avoids duplicate information.

◇ data = data.merge(calendar, on="d", how="left")
' Adds date, events, and holiday information
' by matching on the day column.

' ---------- Merge Price Data ----------

◇ data = data.merge(prices, on=["store_id", "item_id", "wm_yr_wk"], how="left")
' Adds weekly price information for each product.
' Ensures correct price is linked to each sales record.

' ---------- Clean Up Memory ----------

◇ del calendar, prices
' Removes unused datasets from memory.

◇ gc.collect()
' Forces garbage collection to free RAM.

' ---------- Return Final Dataset ----------

◇ return data

' Returns the merged and reshaped dataset.
' This dataset is now ready for feature engineering.

' ---------- Block 6 Summary ----------
' This block:
' - Adds future forecast days
' - Converts sales data to ML-friendly format
' - Merges calendar and price information
' ==

' ==
' BLOCK 7: BASIC FEATURE ENGINEERING
' Purpose:
' - Create simple time-based and price-based features
' - Convert raw date information into numeric form
' - Help the model understand time patterns and price behavior
' ==

◇ def feature_engineering_basic(df):

' This function creates basic features
' from existing columns in the dataset.

' ---------- Convert Day Identifier to Number ----------

◇ df["d_num"] = df["d"].apply(lambda x: x.split("_")[1]).astype(np.int16)

' Converts day labels like "d_1", "d_28" into numbers.
' Example:
' d_28 → 28
' Numeric values are easier for ML models to learn from.

' ---------- Convert Date Column ----------

◇ df["date"] = pd.to_datetime(df["date"])

' Converts date from text format to datetime format.
' This allows extraction of day, month, etc.

' ---------- Extract Day of Week ----------

◆ df["day_of_week"] = df["date"].dt.dayofweek.astype(np.int8)

' Converts date into weekday number.
' Monday = 0, Sunday = 6
' Helps capture weekly sales patterns.

' ---------- Extract Month ----------

◇ df["month"] = df["date"].dt.month.astype(np.int8)
' Extracts month number from date.
' Helps capture seasonal demand patterns.

' ---------- Weekend Indicator ----------

◇ df["is_weekend"] = (df["day_of_week"] >= 5).astype(np.int8)
' Marks Saturday and Sunday as 1.
' Weekends usually have different sales behavior.

' ---------- Price Momentum Feature ----------

◇ df["price_momentum"] = df["sell_price"] / df.groupby("id")["sell_price"].transform("mean")
' Compares current price with average price of the item.
' Value < 1 → discounted price
' Value > 1 → higher-than-average price
' Helps model learn price–demand relationship.

' ---------- Remove Unused Columns ----------

◇ df = df.drop(["date", "d"], axis = 1)
' Removes columns that are no longer needed.
' Keeps dataset clean and compact.

' ---------- Return Updated Dataset ----------

◇ return df
' Returns dataset with basic engineered features.

' ---------- Block 7 Summary ----------
' This block adds simple time and price features
' that help the model learn weekly, monthly,
' and price-based sales patterns.
' ==

' ==
' BLOCK 8: LAG AND ROLLING WINDOW FEATURE ENGINEERING
' Purpose:
' - Capture past sales behavior
' - Help the model learn trends and seasonality
' - Use historical sales to predict future demand
' ==

◇ def feature_engineering_lags(df):
' This function creates lag-based and rolling average features
' from historical sales data.

' ---------- Define Lag Values ----------

◇ lags = [28, 35, 42, 49, 56]
' These values represent past days.
' Example:
' lag_28 → sales 28 days ago
' lag_56 → sales 56 days ago
' Retail demand often repeats weekly and monthly.

' ---------- Create Lag Features ----------

◇ for lag in lags:
' Loops through each lag value.

◇ df[f"lag_{lag}"] = df.groupby("id")["sales"].shift(lag)
' Shifts sales values by given number of days.
' This means:
' Each row gets the sales value from the past.
' Helps the model learn past-to-future relationships.

' ---------- Define Rolling Window Sizes ----------

◇ windows = [7, 14, 28, 60]
' These values represent time windows in days.
' Used to calculate moving averages.

' ---------- Create Rolling Mean Features ----------

◇ for win in windows:
' Loops through each rolling window size.

◇ df[f"rolling_mean_{win}"] = df.groupby("id")["lag_28"].transform(_
lambda x: x.rolling(win).mean())
' Calculates average sales over the given window.
' Example:
' rolling_mean_7 → average sales over last 7 days
' This smooths random fluctuations and noise.

' ---------- Return Updated Dataset ----------

◇ return df

' Returns dataset with lag and rolling features added.

' ---------- Block 8 Summary ----------
' This block creates features that allow the model
' to understand historical trends, seasonality,
' and demand patterns over time.
' ==

' ==
' BLOCK 9: CATEGORICAL VARIABLE ENCODING
' Purpose:
' - Convert text-based columns into numeric values
' - Make data compatible with machine learning models
' - Handle missing event information safely
' ==

◇ from sklearn.preprocessing import LabelEncoder

' Imports LabelEncoder from scikit-learn.
' This tool converts text values into numbers.

◇ def encode_categoricals(df):

' This function encodes all categorical (text) columns
' into numeric form so that ML models can process them.

' ---------- Define Categorical Columns ----------

```
cat_cols = [
    "item_id", "dept_id", "cat_id", "store_id", "state_id",
    "event_name_1", "event_type_1", "event_name_2", "event_type_2"
]
```

' List of columns that contain text values.
' These columns describe products, stores, and events.

' ---------- Handle Missing Event Values ----------

```
df["event_name_1"] = df["event_name_1"].fillna("NoEvent")
df["event_type_1"] = df["event_type_1"].fillna("NoEvent")
df["event_name_2"] = df["event_name_2"].fillna("NoEvent")
df["event_type_2"] = df["event_type_2"].fillna("NoEvent")
```

' Replaces missing event values with "NoEvent".
' This avoids errors during encoding
' and clearly indicates days with no events.

' ---------- Initialize Encoder ----------

◇ encoder = LabelEncoder()
' Creates a LabelEncoder object.
' It assigns a unique number to each unique text value.

' ---------- Encode Each Categorical Column ----------

◇ for col in cat_cols:
' Loops through each categorical column.

◇ df[col] = encoder.fit_transform(df[col].astype(str))
' Converts text values into numeric labels.
' Example:
' CA → 0, TX → 1, WI → 2
' ML models can only work with numbers.

' ---------- Return Encoded Dataset ----------

◇ return df
' Returns dataset with all categorical columns encoded.

' ---------- Block 9 Summary ----------
' This block converts all text-based features
' into numeric form and safely handles missing events,
' making the dataset ready for machine learning.
' ==

' ==
' BLOCK 10: BUILD FINAL PREPROCESSED DATASET
' Purpose:
' - Execute the complete preprocessing pipeline
' - Apply all transformations step by step
' - Save the final ML-ready dataset
' ==

◇ master_df = transform_and_merge(df_sales, df_calendar, df_prices, ProjectConfig)
' Calls the transform_and_merge function.
' - Converts sales data to long format
' - Merges sales, calendar, and price data
' This creates the base dataset for modeling.

◇ master_df = downcast_dtypes(master_df)
' Applies memory optimization again.

' Ensures the merged dataset uses minimum RAM.

◇ master_df = feature_engineering_basic(master_df)
' Adds basic features such as:
' - Day number
' - Day of week
' - Month
' - Weekend indicator
' - Price momentum

◇ master_df = feature_engineering_lags(master_df)
' Adds lag-based and rolling average features.
' Helps the model learn from past sales trends.

◇ master_df = encode_categoricals(master_df)
' Converts all categorical (text) columns into numbers.
' Makes the dataset compatible with ML models.

◇ master_df = downcast_dtypes(master_df)
' Final memory optimization after all features are added.
' Ensures the dataset is efficient and stable.

◇ master_df.to_pickle("processed_dataset.pkl")
' Saves the fully processed dataset to disk.
' This file is directly used for model training.

◇ print(f"Dataset shape: {master_df.shape}")
' Prints the final number of rows and columns.
' Confirms successful preprocessing.

' ---------- Block 10 Summary ----------
' This block runs the entire preprocessing pipeline,
' applies all feature engineering steps,
' and saves the final dataset for machine learning.
' ===

' ===
' BLOCK 11: TRAIN–VALIDATION DATA SPLIT
' Purpose:
' - Split data into training and validation sets
' - Maintain time order (no random shuffling)
' - Prepare inputs and targets for model training
' ===

◇ def perform_split(df, config):
' This function splits the dataset based on time.
' Time-based split is mandatory for forecasting problems.

' ---------- Define Validation Period ----------

◇ valid_mask = (df["d_num"] > (config.TRAIN_END - config.FORECAST_HORIZON)) _
And (df["d_num"] <= config.TRAIN_END)
' Selects the most recent 28 days as validation data.
' This simulates real-world forecasting
' where we predict future using past data.

' ---------- Define Training Period ----------

◇ train_mask = df["d_num"] <= (config.TRAIN_END - config.FORECAST_HORIZON)
' Selects all earlier days as training data.
' Ensures no future data leaks into training.

' ---------- Split Features and Target ----------

◇ X_tr = df[train_mask]
' Training input data.

◇ y_tr = X_tr["sales"]
' Training target variable (actual sales).

◇ X_val = df[valid_mask]
' Validation input data.

◇ y_val = X_val["sales"]
' Validation target variable.

' ---------- Remove Non-Feature Columns ----------

◇ drop_list = ["id", "sales", "wm_yr_wk", "d_num"]
' Columns not used as model features.
' Includes identifiers and target variable.

◇ feats = [c for c in df.columns if c Not In drop_list]
' Final list of feature columns used for training.

' ---------- Return Split Data ----------

◇ return X_tr[feats], y_tr, X_val[feats], y_val, feats

```
' Returns:
' - Training features
' - Training target
' - Validation features
' - Validation target
' - Feature list


' ---------- Block 11 Summary ----------
' This block performs a time-based split
' to ensure fair model evaluation and
' prevent data leakage.
' ============================================================
```