CARLSON SCHOOL
OF MANAGEMENT

Sales Prediction for Walmart

Ching-Hsiao (Tiffany) Chen, Nai-Wen (Amy) Chiu,

Lingyu Liu, Hyemin Yu

MSBA 6421

Table of Content

Project Introduction

---

1. Target Definition

Forecasts play a crucial role in the retail sector, serving as the backbone for supermarkets to shape their strategic growth, make informed tactical choices, and efficiently navigate demand and supply planning. Accurate forecasting is essential to prevent customer service disruptions and reduce excessive inventory expenses. Overstocking results in additional costs, while understocking can cause missed sales opportunities and reduced profit margins. In this context, our objective is to assist Walmart with predicting daily sales in the USA for the upcoming 28 days.

2. Expected Contribution on Business

Accurate forecasting benefits Walmart's operation in several aspects:

- Inventory Management: Reduce overstock and stockouts, thereby lowering storage costs and preventing lost sales.
- Resource Allocation: Ensuring optimal use of resources, including workforce and supply chain logistics, and have enough preparation before significant sales campaigns
- Strategic Planning: Supporting decisions for long-term business strategies, like expansion or promotional activities.
- Financial Performance: Improving sales predictions directly impacts revenue and profit margins, aiding in more precise financial planning

3. Data Introduction

The hierarchical dataset involves the unit sales of 3,049 products, classified in 3 product categories (Hobbies, Foods, and Household) and 7 product departments. The products are sold across ten stores, located in three States (CA, TX, and WI). The historical data range from **2011-01-29** to **2016-06-19**.

The M5 Accuracy dataset consists of 3 files:

1. "*calendar.csv*"

- *date*: The date in a "y-m-d" format.
- *wm_yr_wk*: The id of the week the date belongs to.
- *weekday*: The type of the day (Saturday, Sunday, …, Friday).
- *wday*: The id of the weekday, starting from Saturday.
- *month*: The month of the date.
- *year*: The year of the date.
- *event_name_1*: If the date includes an event, the name of this event.
- *event_type_1*: If the date includes an event, the type of this event.
- *event_name_2*: If the date includes a second event, the name of this event.
- *event_type_2*: If the date includes a second event, the type of this event.
- *snap_CA*, *snap_TX*, and *snap_WI*: A binary variable (0 or 1) indicating whether the stores of CA, TX or WI allow SNAP(Supplement Nutrition Assistance Program) purchases on the examined date. 1 indicates that SNAP purchases are allowed.

2: *"sell_prices.csv"*

- *store_id*: The id of the store where the product is sold.

- *item_id*: The id of the product.
- *wm_yr_wk*: The id of the week.
- *sell_price*: The price of the product for the given week/store. The price is provided per week (average across seven days). If not available, this means that the product was not sold during the examined week. Note that although prices are constant at weekly basis, they may change through time (both training and test set).

3: "*sales_train.csv*"

- *item_id*: The id of the product.
- *dept_id*: The id of the department the product belongs to.
- *cat_id*: The id of the category the product belongs to.
- *store_id*: The id of the store where the product is sold.
- *state_id*: The State where the store is located.
- *d_1, d_2, ..., d_i, ... d_1941*: The number of units sold at day *i*, starting from 2011-01-29.

Technical Solution Overview

---

1. Tools and Packages

The main environment of this project was a Jupyter Notebook on Kaggle, with Python **3.10.12**.

Such packages are also used as below.

- Pandas (v.)
- Numpy (v.)
- Matplotlib (v.)
- Seaborn (v.)
- Keras (v.)
- Tensorflow (v.)
- Lightgbm (v.)

2. Technical Structure of our Solution

Given that we are dealing with the prediction of daily sales based on historical sales data, we have to take the dynamic sales patterns over time into consideration. We employ the following techniques as our main structure of modeling to achieve accurate forecasts.

1. Time Series: Decompose the time series into its components of trend and seasonality. This helps with understanding the underlying patterns and capturing the cyclic nature of sales.
2. Model Selection Strategy: Explore models including LightGBM and LSTM networks for dealing with more intricate relationships within the data. In addition, we consider creating an ensemble of multiple models to ensure more consistent performance across various scenarios.

Predictive Model 1: LightGBM (Light Gradient-Boosting Machine)

---

1. Methodology

Addressing time series prediction challenges necessitates a comprehensive understanding of the trends and patterns inherent in states, categories, or departments over time. Consequently, we first initiated a thorough trends analysis. Subsequently, leveraging insights from this analysis, we undertook feature engineering to capture the sales trends over time. Recognizing the variability of sales trends across products, locations, and time spans, we developed distinct LightGBM (Light Gradient-Boosting Machine) models for each category and department to capture sales patterns within different states and time spans. Finally, we aggregated the prediction results from these models to formulate the conclusive prediction.

2. Data-Preprocessing

2.1 Data Loading and Merging

Three datasets are first downsized to reduce memory usage and computation cost.

```python
#Downcast in order to save memory
def downcast(df):
    cols = df.dtypes.index.tolist()
    types = df.dtypes.values.tolist()
    for i,t in enumerate(types):
        if 'int' in str(t):
            if df[cols[i]].min() > np.iinfo(np.int8).min and df[cols[i]].max() < np.iinfo(np.int8).max:
                df[cols[i]] = df[cols[i]].astype(np.int8)
            elif df[cols[i]].min() > np.iinfo(np.int16).min and df[cols[i]].max() < np.iinfo(np.int16).max:
                df[cols[i]] = df[cols[i]].astype(np.int16)
            elif df[cols[i]].min() > np.iinfo(np.int32).min and df[cols[i]].max() < np.iinfo(np.int32).max:
                df[cols[i]] = df[cols[i]].astype(np.int32)
            else:
                df[cols[i]] = df[cols[i]].astype(np.int64)
        elif 'float' in str(t):
            if df[cols[i]].min() > np.finfo(np.float16).min and df[cols[i]].max() < np.finfo(np.float16).max:
                df[cols[i]] = df[cols[i]].astype(np.float16)
            elif df[cols[i]].min() > np.finfo(np.float32).min and df[cols[i]].max() < np.finfo(np.float32).max:
                df[cols[i]] = df[cols[i]].astype(np.float32)
            else:
                df[cols[i]] = df[cols[i]].astype(np.float64)
        elif t == object:
            if cols[i] == 'date':
                df[cols[i]] = pd.to_datetime(df[cols[i]], format='%Y-%m-%d')
            else:
                df[cols[i]] = df[cols[i]].astype('category')
    return df
```

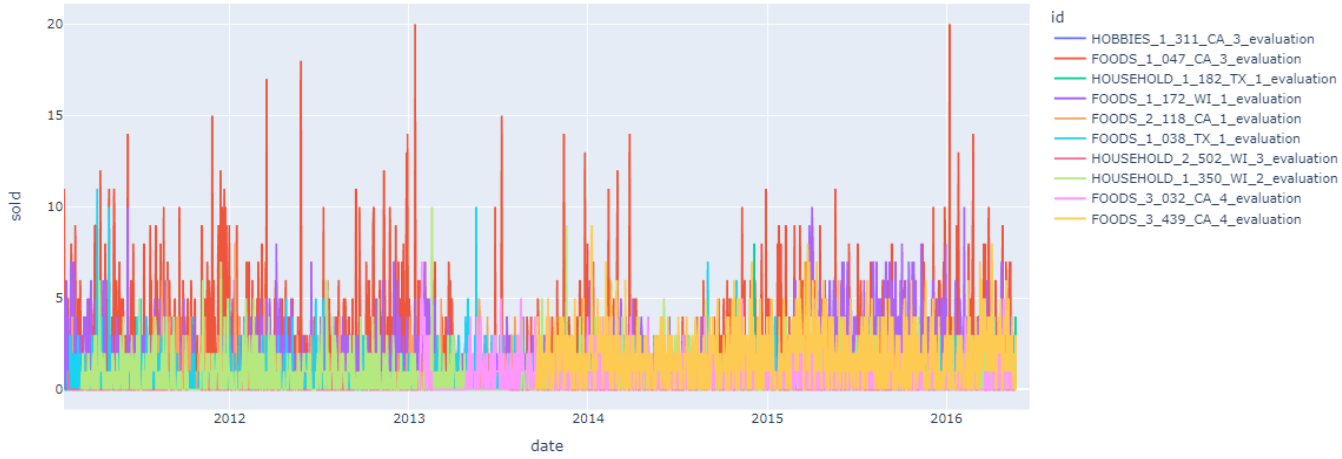
The sales_train dataset initially adopts a wide format. We subsequently transformed into a long format by melting the data. Following this, we merged three datasets into a single table using the "id" column as the key identifier.

While merging the tables, we observed zero sales in the sales_train dataset, attributed to items not yet released. To optimize memory usage, we pruned the dataset by excluding rows where the item sales dates preceded their respective product release.

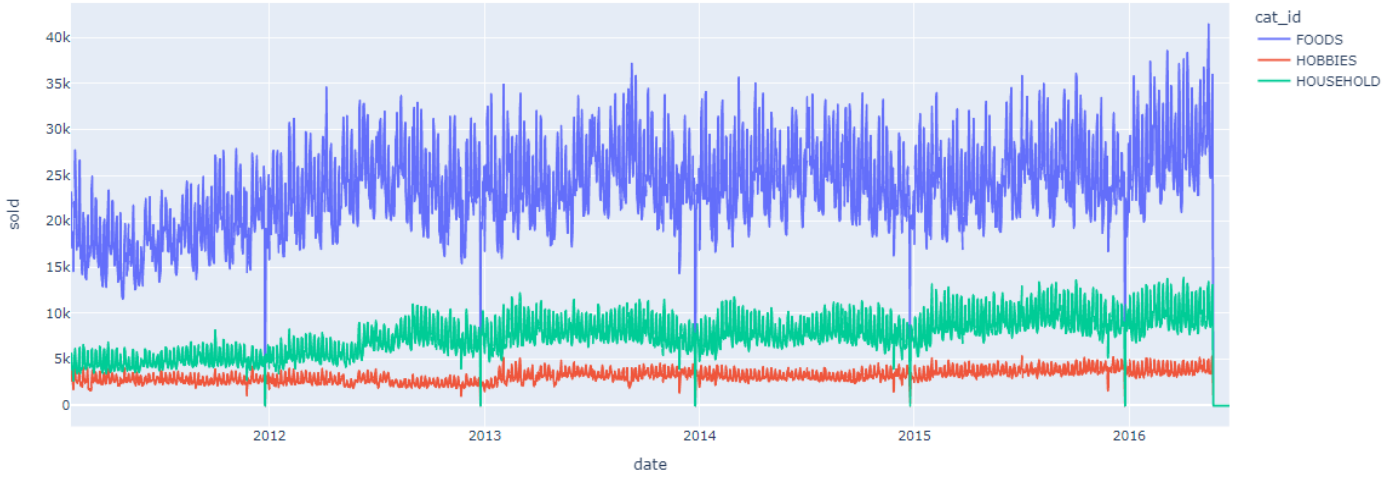
2.2 Data Exploratory

After merging datasets, we explored the data to get a better understanding of the sales patterns. This guided us in figuring out the best ways to build our models and identifying types of features we need to create.

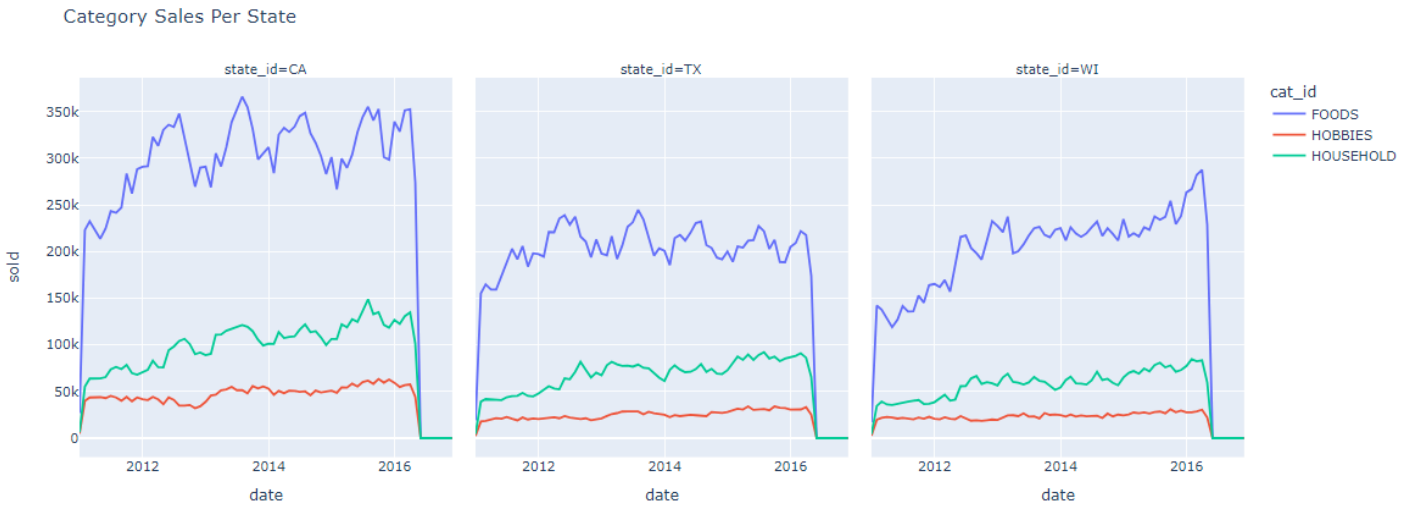
Upon randomly selecting 10 items, a persistent trend of consistently low sales per item became apparent, with no significant spikes observed. Upon further analysis, it became evident that discernible trends or seasonality were lacking, suggesting significant noise within the dataset.



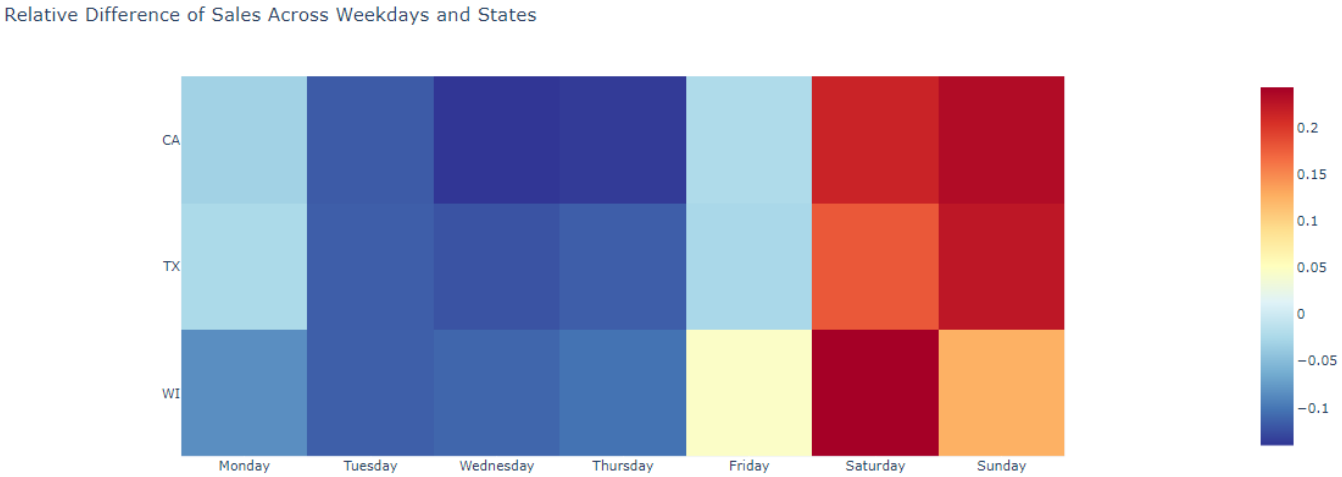
However, examining the data at a higher category level, notable sales patterns emerged. Overall sales exhibited upward trends, highlighting distinct patterns among various categories. This observation underscores the necessity of developing separate models for each item category. In a closer inspection, robust weekly and monthly seasonality patterns also became apparent.



Monthly sales by state and category also affirmed the need for distinct models, given the observed variations and patterns.



While examining the impact of days of week in sales, we observed relatively higher sales on weekends compared to weekdays. This emphasizes the importance of incorporating time-based features into our models.



3. Feature Engineering

Building on the insights gained from Exploratory Data Analysis (EDA), we incorporated new features into our models to enhance their predictive capabilities.

3.1 Time-based Features

Temporal features, including day of the week, month, quarter, and year, were extracted from the timestamp to capture seasonal patterns. Additionally, lag features were created by shifting the time series data, representing values at previous time steps to capture past sales patterns. In total, 15 days of lag features were included.

The lags features are:

- sold_lag_1
- sold_lag_2
- sold_lag_3
- sold_lag_4

  ⋮

- sold_lag_15

```
#Introduce lags
lags = np.arange(0, 15, 1)

for lag in lags:
    df['sold_lag_'+str(lag)] = df.groupby(['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id'],as_index=False)['sold'].shift(lag).astype(np.float16)
```

3.2 Rolling Statistic

We generated rolling mean, rolling standard deviation, the difference between rolling means, and rolling maximum of sales. These features serve to smooth short-term fluctuations and emphasize long-term trends. Utilizing rolling windows of 7, 14, 30, 60, and 180, we incorporated these variations into our analysis.

The rolling statistic features are:

- rm_7, rm_14, rm_30, rm_60, rm_180
- std_7, std_14, std_30, std_60, std_180
- diff_rm_7, diff_rm_14, diff_rm_30, diff_rm_60, diff_rm_180
- max_7, max_14, max_30, max_60, max_180

```
for roll in [7, 14, 30, 60, 180]:
  df[f'rm_{roll}'] = df.groupby(['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id'])['sold'].transform(lambda x: x.rolling(roll).mean())
  df[f'std_{roll}'] = df.groupby(['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id'])['sold'].transform(lambda x: x.rolling(roll).std())
  df[f'diff_rm_{roll}'] = df.groupby(['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id'])['sold'].transform(lambda x : x.diff().rolling(roll).mean())
  df[f'max_{roll}'] = df.groupby(['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id'])['sold'].transform(lambda x: x.rolling(roll).max())
  df = downcast(df)
```

3.3 Price

We extended the application of a similar methodology of rolling statistics to the price column, wherein we computed the maximum, minimum, standard deviation, mean, normalized value, as well as monthly and yearly momentum of prices categorized by store_id and item_id. These features captured variations in prices across stores and items. The normalization of prices and the

derived statistical metrics further elucidated the relative pricing dynamics over distinct time intervals.

The price statistic features are:

- price_max
- price_min
- price_std
- price_mean
- prev_sell_price
- price_norm
- price_momentum
- price_momentum_m
- price_momentum_y

```
df['price_max'] = df.groupby(['store_id','item_id'])['sell_price'].transform('max')
df['price_min'] = df.groupby(['store_id','item_id'])['sell_price'].transform('min')
df['price_std'] = df.groupby(['store_id','item_id'])['sell_price'].transform('std')
df['price_mean'] = df.groupby(['store_id','item_id'])['sell_price'].transform('mean')
df['prev_sell_price'] = df.groupby(['store_id','item_id'])['sell_price'].transform(lambda x: x.shift(1))

gc.collect()

df['price_norm'] = df['sell_price']/df['price_max']
df['price_momentum'] = df['sell_price']/df.groupby(['store_id','item_id'])['sell_price'].transform(lambda x: x.shift(1))
df['price_momentum_m'] = df['sell_price']/df.groupby(['store_id','item_id','month'])['sell_price'].transform('mean')
df['price_momentum_y'] = df['sell_price']/df.groupby(['store_id','item_id','year'])['sell_price'].transform('mean')
```

3.4 Time Since Last Event

The days preceding or following SNAP events can significantly impact sales. To quantify the descending or ascending strength of this influence, we have introduced columns indicating the number of days since the last SNAP event.

The time since last events feature is:

- snap_days_since_last

```
df['snap_CA_days_since_last'] = np.nan

ca = df.loc[df['state_id'] == 'CA'][['d', 'snap_CA']]
ca['diff'] = ca['d'].where(ca['snap_CA'] == 1).ffill()
ca['diff'] = ca['d'] - ca['diff']

df['snap_CA_days_since_last'].loc[df['state_id'] == 'CA'] = ca['diff']
```

(For simplicity, we omitted repeated calculations.)

3.5 Mean Encoding

We employed mean-encoding to address categorical columns, including item_id, dept_id, cat_id, and state_id. Recognizing the relevance of these columns to sales, we applied mean encoding to transform categorical data into numerical representations. This technique involves replacing each category with the mean sales value associated with that specific category.

The meaning encoding features are:

- item_sold_avg
- state_sold_avg
- store_sold_avg
- cat_sold_avg
- dept_sold_avg
- state_cat_sold_avg
- state_dept_sold_avg
- cat_dept_sold_avg
- store_cat_sold_avg
- store_dept_sold_avg
- store_item_sold_avg
- cat_item_sold_avg
- dept_item_sold_avg
- state_store_sold_avg

- state_store_cat_sold_avg
- store_cat_dept_sold_avg

```
df['item_sold_avg'] = df.groupby('item_id')['sold'].transform('mean').astype(np.float16)
df['state_sold_avg'] = df.groupby('state_id')['sold'].transform('mean').astype(np.float16)
df['store_sold_avg'] = df.groupby('store_id')['sold'].transform('mean').astype(np.float16)
df['cat_sold_avg'] = df.groupby('cat_id')['sold'].transform('mean').astype(np.float16)
df['dept_sold_avg'] = df.groupby('dept_id')['sold'].transform('mean').astype(np.float16)
df['cat_dept_sold_avg'] = df.groupby(['cat_id','dept_id'])['sold'].transform('mean').astype(np.float16)
df['store_item_sold_avg'] = df.groupby(['store_id','item_id'])['sold'].transform('mean').astype(np.float16)
df['cat_item_sold_avg'] = df.groupby(['cat_id','item_id'])['sold'].transform('mean').astype(np.float16)
df['dept_item_sold_avg'] = df.groupby(['dept_id','item_id'])['sold'].transform('mean').astype(np.float16)
df['state_store_sold_avg'] = df.groupby(['state_id','store_id'])['sold'].transform('mean').astype(np.float16)
df['state_store_cat_sold_avg'] = df.groupby(['state_id','store_id','cat_id'])['sold'].transform('mean').astype(np.float16)
df['store_cat_dept_sold_avg'] = df.groupby(['store_id','cat_id','dept_id'])['sold'].transform('mean').astype(np.float16)
```

4. Model Building

4.1 LightGBM (Light Gradient-Boosting Machine)

We used LightGBM as the base model. LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with several advantages such as faster training with higher efficiency, lower memory usage, better accuracy and support of parallel, distributed and GPU learning, etcetera[1].

Through the application of Bayes Optimization and iterative experimentation, we identified the optimal hyperparameter settings:

- 'boosting_type': 'gbdt',
- 'objective': 'tweedie',
- 'tweedie_variance_power': 1.1,
- 'metric': 'rmse',
- 'subsample': 0.5,
- 'subsample_freq': 1,
- 'learning_rate': 0.015,
- 'num_leaves': 2**8-1,
- 'min_data_in_leaf': 2**8-1,
- 'feature_fraction': 0.5,
- 'max_bin': 100,

- 'n_estimators': 1000,
- 'boost_from_average': False,
- 'verbose': -1,
- 'seed' : 1995

To prevent overfitting and reduce computational costs, we implemented early stopping with a parameter setting of stopping_rounds=50 during the training process. Further details on training and predicting will be explored in the next section.

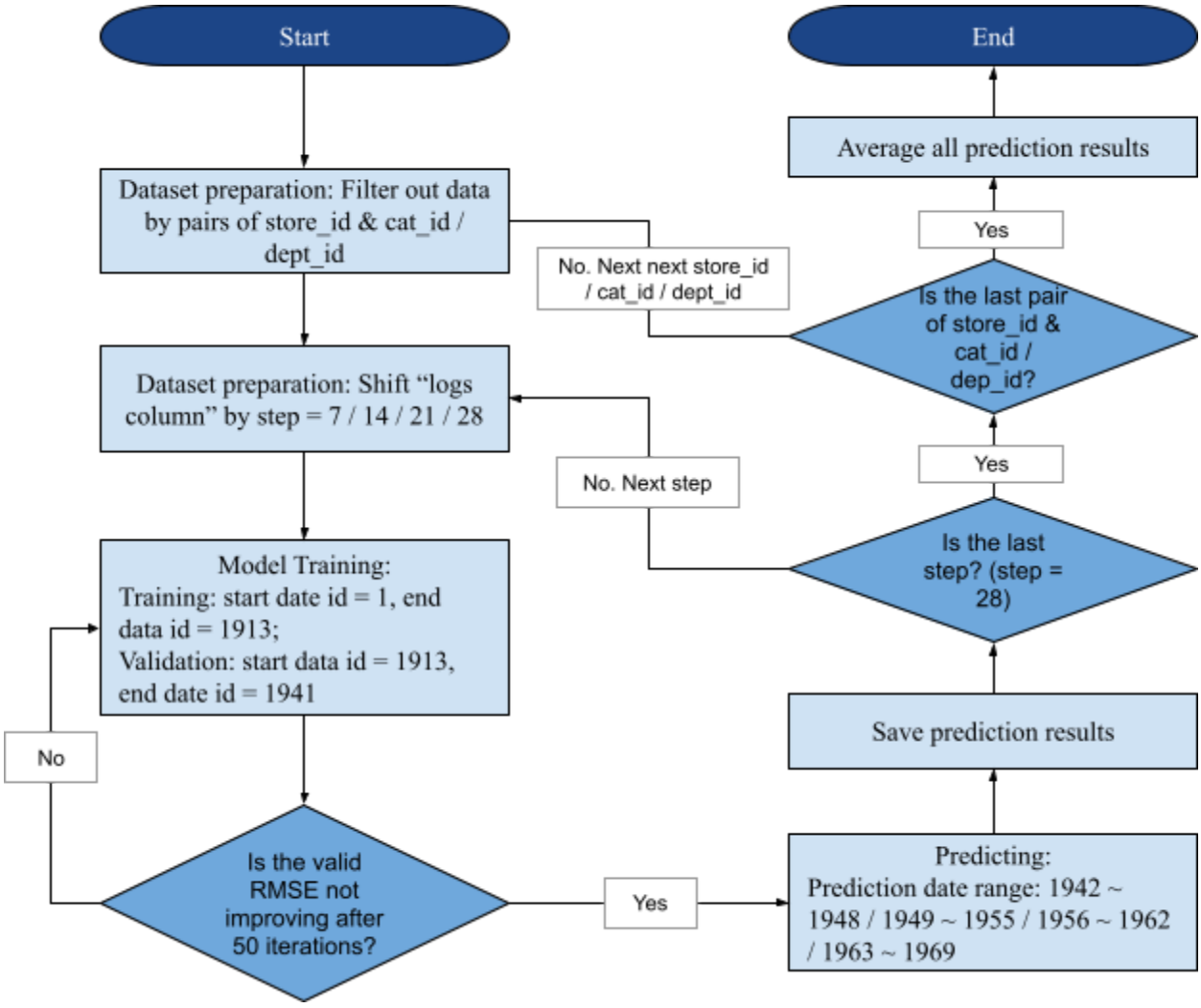
4.2 Training and Predicting Process

To address sales variations across states, categories, and departments, we created models for every distinct pairing of states and categories, as well as states and departments.

Next, we employed the same dataset for training models and forecasting sales using various step sizes. We adjusted the lag columns according to the specified step. For example, when predicting with a step size of 7, we shifted the lag columns by 7, 14, 21, and 28 in each iteration. This approach enables the utilization of the most recent historical information for prediction, instead of shifting the entire model to accommodate the 28-day period.

After generating predictions using different models, we ensemble the results by concatenating them together and averaging them. We tried step = 2, step = 4, step = 7, with Weighted Root Mean Squared Scaled Error (RMSSE) 0.53199, 0.64806, and 0.54488 respectively.

Using a step size of 7 as an example, our model training and prediction process is as follow:


Start
Dataset preparation: Filter out data by pairs of store_id & cat_id / dept_id
Dataset preparation: Shift "logs column" by step = 7 / 14 / 21 / 28
Model Training:
Training: start date id = 1, end data id = 1913;
Validation: start data id = 1913, end date id = 1941
Is the valid RMSE not improving after 50 iterations?
No
Yes
Predicting:
Prediction date range: 1942 ~ 1948 / 1949 ~ 1955 / 1956 ~ 1962 / 1963 ~ 1969
Save prediction results
Is the last step? (step = 28)
No. Next step
Yes
Is the last pair of store_id & cat_id / dep_id?
No. Next next store_id / cat_id / dept_id
Yes
Average all prediction results
End

```
predictions = pd.DataFrame()

for store in STORES:
    for state in CATS:
        for step in STEPS:
          print(store,state, 'start')
          grid_df = prepare_data(store, state)
          grid_df[lags_col] = grid_df.groupby(['id'], observed=False)[lags_col].shift(step)
          model_var = grid_df.columns[~grid_df.columns.isin(remove_feature)]
          # ix_to_drop = grid_df[(grid_df['d'] <= 1941) & grid_df.isna().any(axis=1)].index
          # grid_df.drop(index=ix_to_drop, inplace=True)

          pred_start = FIRST_PRED_DAY + step - VAL_DAYS
          pred_end = FIRST_PRED_DAY + step - 1


          tr_mask = (grid_df['d'] >= TRAIN_START) & (grid_df['d'] <= TRAIN_END)
          vl_mask = (grid_df['d'] >= VAL_START) & (grid_df['d'] <= VAL_END)
          pr_mask = (grid_df['d'] >= pred_start) & (grid_df['d'] <= pred_end)

          trainX = grid_df[tr_mask][model_var]
          trainY = grid_df[tr_mask][TARGET]
          valX = grid_df[vl_mask][model_var]
          valY = grid_df[vl_mask][TARGET]
          testX = grid_df[pr_mask][model_var]
          print(f'Train shape: {trainX.shape}. Val shape: {valX.shape}. Test shape: {testX.shape}')
```

```
          # Train
          lgbm = lgb.LGBMRegressor(**lgb_params)
          callbacks = [early_stopping(stopping_rounds=50, first_metric_only=False)]

          lgbm.fit(trainX, trainY,
                  eval_set=[(valX, valY)],
                  eval_metric='rmse',
                  callbacks=callbacks)

          # Predict
          yhat = lgbm.predict(testX, num_iteration=lgbm.best_iteration_)
          preds = grid_df[(grid_df['d'] >= pred_start) & (grid_df['d'] <= pred_end)][['id', 'd']]
          preds['sales'] = yhat
          predictions = pd.concat([predictions, preds], axis=0)
          predictions.to_pickle(f'{submission_dir}before_ensemble/me_with_steps_store_cat_preds_temp_2_1_missing.pkl')

          del grid_df, trainX, trainY, valX, valY, testX, lgbm, tr_mask, vl_mask, pr_mask ; gc.collect
```

(For simplicity, we removed the training and predicting code for store_id - dept_id pairs, as the only distinction lies in the second for loop.)

Predictive Model 2: LSTM (Long Short-Term Memory) Network

---

1. Methodology

Long Short-Term Memory Network is designed to overcome the limitations of traditional RNNs in capturing and learning long-term dependencies in sequential data. LSTMs are widely used in time based prediction. In the process of our LSTM network, we defined a specific time range starting day based on our trend analysis. Subsequently, we created a feature to include the impact of events. As for the training and testing data split, we divided our dataset based on time span and starting day. After having our input data ready, we tried different hidden layers and parameter settings to optimize the model performance. Finally, we used our defined time range to forecast the sales of the first unknown upcoming day. In general, This approach ensures a comprehensive and informed forecasting methodology.

2. Data-Preprocessing

2.1 Data Loading and Merging

Two datasets are first downsized to reduce memory usage and computation cost.

```
sales = pd.read_csv('/kaggle/input/m5-forecasting-accuracy/sales_train_evaluation.csv')
sales.name = 'sales'
calendar = pd.read_csv('/kaggle/input/m5-forecasting-accuracy/calendar.csv')
calendar.name = 'calendar'
```

```
import numpy as np
#Downcast in order to save memory
def downcast(df):
    cols = df.dtypes.index.tolist()
    types = df.dtypes.values.tolist()
    for i,t in enumerate(types):
        if 'int' in str(t):
            if df[cols[i]].min() > np.iinfo(np.int8).min and df[cols[i]].max() < np.iinfo(np.int8).max:
                df[cols[i]] = df[cols[i]].astype(np.int8)
            elif df[cols[i]].min() > np.iinfo(np.int16).min and df[cols[i]].max() < np.iinfo(np.int16).max:
                df[cols[i]] = df[cols[i]].astype(np.int16)
            elif df[cols[i]].min() > np.iinfo(np.int32).min and df[cols[i]].max() < np.iinfo(np.int32).max:
                df[cols[i]] = df[cols[i]].astype(np.int32)
            else:
                df[cols[i]] = df[cols[i]].astype(np.int64)
        elif 'float' in str(t):
            if df[cols[i]].min() > np.finfo(np.float16).min and df[cols[i]].max() < np.finfo(np.float16).max:
                df[cols[i]] = df[cols[i]].astype(np.float16)
            elif df[cols[i]].min() > np.finfo(np.float32).min and df[cols[i]].max() < np.finfo(np.float32).max:
                df[cols[i]] = df[cols[i]].astype(np.float32)
            else:
                df[cols[i]] = df[cols[i]].astype(np.float64)
        elif t == object:
            if cols[i] == 'date':
                df[cols[i]] = pd.to_datetime(df[cols[i]], format='%Y-%m-%d')
            else:
                df[cols[i]] = df[cols[i]].astype('category')
    return df

sales = downcast(sales)
```

3. Feature Engineering

3.1 Transpose a Dataframe

First, we swapped the rows and columns of the sales dataset to structure the DataFrame with one row for each day.

```
df = sales.T
df.head(8)
```

	0	1	2	3
id	HOBBIES_1_001_CA_1_evaluation	HOBBIES_1_002_CA_1_evaluation	HOBBIES_1_003_CA_1_evaluation	HOBBIES_1_004_CA_1_evaluation
item_id	HOBBIES_1_001	HOBBIES_1_002	HOBBIES_1_003	HOBBIES_1_004
dept_id	HOBBIES_1	HOBBIES_1	HOBBIES_1	HOBBIES_1
cat_id	HOBBIES	HOBBIES	HOBBIES	HOBBIES
store_id	CA_1	CA_1	CA_1	CA_1
state_id	CA	CA	CA	CA
d_1	0	0	0	0
d_2	0	0	0	0

3.2 Set Up Time Stamp and Start Day

Based on trend analysis, the first year exhibits relatively lower sales in contrast to subsequent years. Consequently, we've configured the start day to be 350 days.

```
timesteps = 7
startDay = 350
# drop the categorical rows
df = df[6 + startDay:]
```

3.3 Event-based Features

Created feature to include impact of event.

```
daysBeforeEvent = pd.DataFrame(np.zeros((1969,1)))
```

```
import datetime as dt

#if first day was an event this row will cause an exception because "x-1".
#Since it is not i did not consider for now
for i,j in calendar.iterrows():
    if((pd.isnull(calendar["event_name_1"][i]) and pd.isnull(calendar["event_name_2"][i])) == False):
           daysBeforeEvent[0][i-1] = 1


daysBeforeEventTest = daysBeforeEvent[1941:1969]
daysBeforeEvent = daysBeforeEvent[startDay:1941]

daysBeforeEvent.columns = ["oneDayBeforeEvent"]
daysBeforeEvent.index = df.index

df = pd.concat([df, daysBeforeEvent], axis = 1)
```

3.4 Normalization

Applied Min Max Scaler Normalization on the dataset.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range = (0, 1))
df_scaled = scaler.fit_transform(df)
```

3.5 Train-test Split

Our dataset is divided into training and testing data, taking into account the start day and time range.

```
X_train = []
y_train = []
for i in range(timesteps, 1941 - startDay):
    X_train.append(df_scaled[i-timesteps:i])
    y_train.append(df_scaled[i][0:30490])
```

4. Model Building

LSTM helps mitigate the vanishing gradient problem, which is a common issue in training deep networks, enabling the model to capture long-term dependencies in sequential data.

We experimented with various model settings and ultimately determined that an LSTM network with three hidden layers exhibited the most decent performance.

```
input_unit_size = 1584*7

# Importing the Keras libraries and packages
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout


# Initialising the model setting
model = Sequential()

# Adding the 1st LSTM layer and some Dropout regularisation
model.add(LSTM(units = 550, return_sequences = True, input_dim = input_unit_size,activation='relu',
                   input_shape = input_shape))
model.add(Dropout(0.25))
# Adding a 2nd LSTM layer and some Dropout regularisation
model.add(LSTM(units = 400, return_sequences = True, activation='relu'))
model.add(Dropout(0.25))

# Adding a 3rd LSTM layer and some Dropout regularisation
model.add(LSTM(units = 400, return_sequences = False, activation='relu'))

# Adding the output layer
model.add(Dense(units = 30490))

# Compiling
model.compile(optimizer = 'adam', loss = 'mse', metrics=['mean_squared_error'])
```

```
# Fit model ans stack
model.fit(X_train, y_train, epochs = 150, batch_size = 128, verbose=2)
```

```
Epoch 1/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 163ms/step
Epoch 2/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 137ms/step
Epoch 3/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 138ms/step
Epoch 4/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 138ms/step
Epoch 5/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 137ms/step
Epoch 6/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 139ms/step
Epoch 7/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 140ms/step
Epoch 8/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 138ms/step
Epoch 9/150
13/13 - 2s - loss: 0.0095 - mean_squared_error: 0.0095 - 2s/epoch - 137ms/step
Epoch 10/150
13/13 - 2s - loss: 0.0094 - mean_squared_error: 0.0094 - 2s/epoch - 136ms/step
Epoch 11/150
13/13 - 2s - loss: 0.0094 - mean_squared_error: 0.0094 - 2s/epoch - 140ms/step
Epoch 12/150
```

Finally, we used the past 7 days, which is our defined time range, to forecast the sales of the first unknown upcoming day.

```
inputs= df[-timesteps:]
inputs = scaler.transform(inputs)
```

```
X_test = []
X_test.append(inputs[0:14])
X_test = np.array(X_test)
predictions = []

for j in range(timesteps,timesteps + 28):
    y_predict = model.predict(X_test[0,j - timesteps:j].reshape(1, timesteps, 30491))
    testappend = np.column_stack((np.array(y_predict), daysBeforeEventTest[0][1941 + j - timesteps]))
    X_test = np.append(X_test, testappend).reshape(1,j + 1,30491)
    y_predict = scaler.inverse_transform(testappend)[:,0:30490]
    predictions.append(y_predict)
```

<table>
<tr><td>D1908</td><td>D1909</td><td>D1910</td><td>D1911</td><td>D1912</td><td>D1913</td><td>D1914</td><td>D1915</td><td>D1916</td><td>D1917</td></tr>
<tr><td colspan="7">Use these 7 days for prediction</td><td>Predicted value</td><td></td><td></td></tr>
<tr><td></td><td colspan="7">Use these 7 days for prediction</td><td>Predicted value</td><td></td></tr>
<tr><td></td><td></td><td colspan="7">Use these 7 days for prediction</td><td>Predicted value</td></tr>
</table>

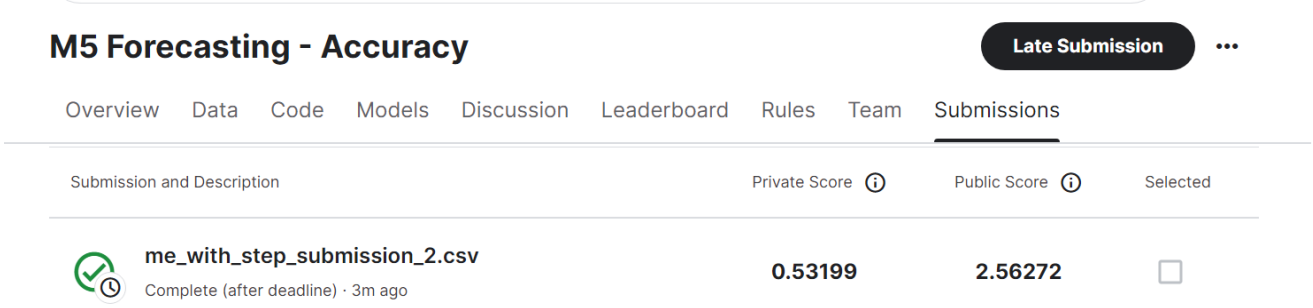
Model Results and Selection

During the training process of the two distinct main models, Root Mean Squared Error (RMSE) was employed as the performance metric. This choice was made due to uncertainties regarding the specific scaler for each prediction in calculating Root Mean Squared Scaled Error (RMSSE). However, for the comparative analysis of various modeling methods, we utilized the Weighted Root Mean Squared Scaled Error (WRMSSE) provided by each Kaggle submission.

The top score for each model are as follow:

Algorithms	Top Private Score (WRMSSE)
LightGBM	0.53199
LSTM	0.6919

Based on the scores, we have chosen LightGBM with a step size of 2 as our final model. The prediction results and business insights will be presented in the next section.



Feature Summary

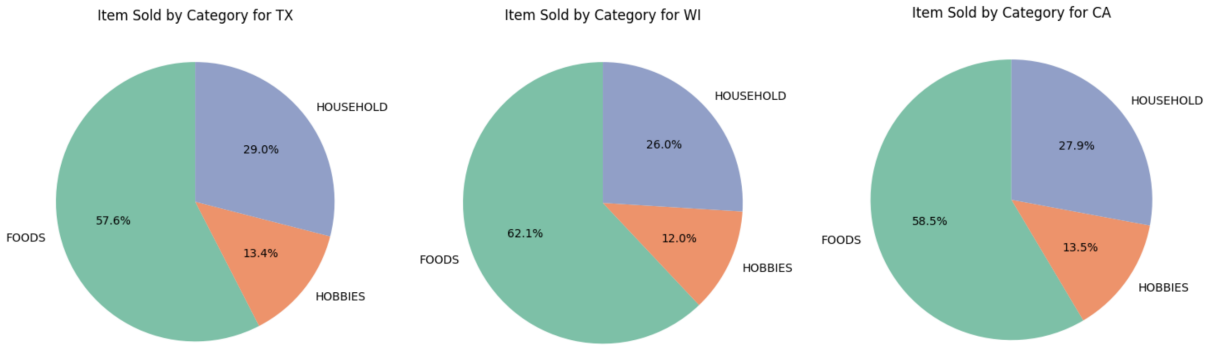
Algorithms	Features
LightGBM	sold_lag_1~15
	rm_7, rm_14, rm_30, rm_60, rm_180
	std_7, std_14, std_30, std_60, std_180
	diff_rm_7, diff_rm_14, diff_rm_30, diff_rm_60, diff_rm_180
	max_7, max_14, max_30, max_60, max_180
	price_max
	price_min
	price_std
	price_mean
	prev_sell_price
	price_norm
	price_momentum
	price_momentum_m
	price_momentum_y
	snap_days_since_last
	item_sold_avg
	state_sold_avg
	store_sold_avg
	cat_sold_avg
	dept_sold_avg
	state_cat_sold_avg
	state_dept_sold_avg
	cat_dept_sold_avg

	store_cat_sold_avg
	store_dept_sold_avg
	store_item_sold_avg
	cat_item_sold_avg
	dept_item_sold_avg
	state_store_sold_avg
	state_store_cat_sold_avg
	store_cat_dept_sold_avg
LSTM	d_1~d_1941
	event_name_1
	event_name_2

Business Value and Recommendation

We utilized the predictive outcomes from LightGBM to derive valuable insights.

From the pie chart of items sold across categories in each state, we noticed a similar pattern in each state. The Food category stands out with the highest number of units sold. As a result, Walmart should emphasize inventory management for the Foods category, while also considering potential marketing campaigns to boost sales in the Hobbies category.

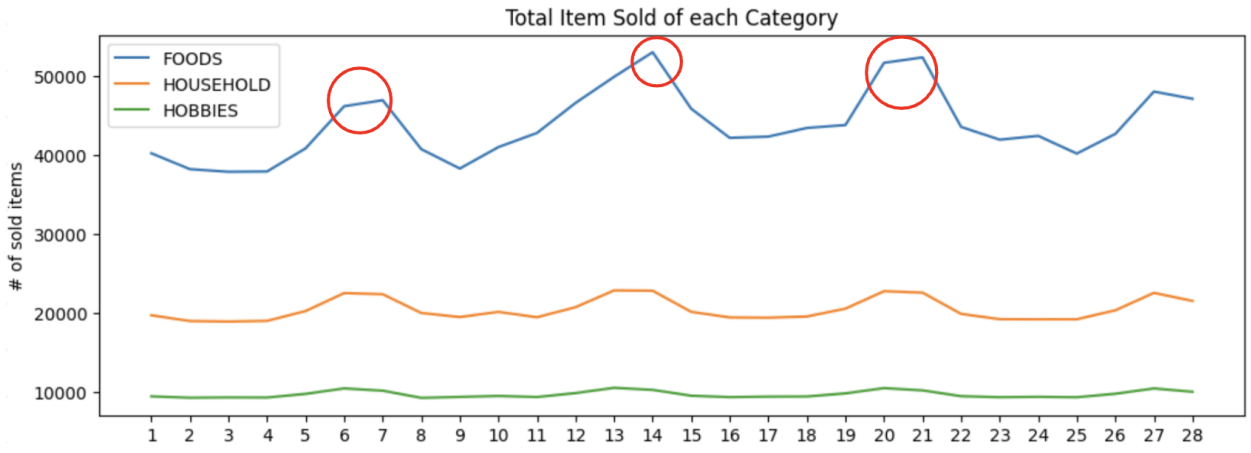


We also created a table detailing the sales units across stores between each state, helping us in pinpointing the stores with the highest unit sales. This information will be invaluable for Walmart to optimize logistical and supply chain arrangements effectively.

State	Store (Order by # of sold)	Percentage
CA	CA_3	30.53%
	CA_2	25.45%
	CA_1	25.24%
	CA_4	18.78%
TX	TX_2	34.56%
	TX_3	34.19%
	TX_1	31.25%

WI	WI_2	37.47%
	WI_1	31.29%
	WI_3	31.24%

Finally, we've generated a line chart to depict the total items sold across various categories in the upcoming 28 days. Notably, there are significant peaks in the food category at the end of the first, second, and third weeks. Therefore, we recommend that Walmart allocates extra resources in preparation for the upcoming peak days.



Appendix

---

1. https://lightgbm.readthedocs.io/en/stable/

2.https://github.com/Mcompetitions/M5-methods/tree/master/Code%20of%20Winning%20Methods/A1

3.https://www.kaggle.com/code/headsortails/back-to-predict-the-future-interactive-m5-eda

4.https://www.kaggle.com/code/anshuls235/time-series-forecasting-eda-fe-modelling

5.https://www.analyticsvidhya.com/blog/2019/12/6-powerful-feature-engineering-techniques-time-series/

6.https://www.kaggle.com/c/m5-forecasting-accuracy/discussion/151927

7.https://robjhyndman.com/papers/rectify.pdf

8.https://medium.com/@soumyachess1496/cross-validation-in-time-series-566ae4981ce4

9.https://www.kaggle.com/code/anshuls235/time-series-forecasting-eda-fe-modelling#6.-Modelling-and-Prediction

10.https://www.kaggle.com/code/tarunpaparaju/m5-competition-eda-models/notebook#Modeling-