



PREDICTION ON WALMART SALES DEMAND

Data from Kaggle M5 Competition

MSBA

Table of Contents

I. Project Definition & Business Objectives	3
1.1 What Is the Business Problem?	3
1.2 What Is the Objective of this Project?	3
II. Technical Specifications & Solution Overview	4
2.1 Tools	4
2.2 Datasets	4
2.3 Solution Overview	4
III. Predictive Model 1 - Ensembling Model with LightGBM	6
3.1 Methodology	6
3.2 Data preparation	6
3.3 Feature engineering	7
3.5 Parameter Tuning	10
IV. Predictive Model 2 - SARIMAX	13
4.1 Methodology	13
4.2 Data preparation	13
4.2.1 Training data	13
4.2.2 Prediction data	14
4.3 Modeling	15
V. Predictive Model 3 - Prophet	16
5.1 Methodology	16
5.2 Data preparation	16
5.2.1 Training data	16
5.2.2 Prediction data	17
5.3 Modeling	17
VI. Predictive Model 4 - Temporal Fusion Transformer	21

6.1 Methodology	21
6.2 Data Preparation	22
6.3 Modeling	24
6.4 Hyperparameter Tuning	25
VII. Predictive Model 5 - Seq2Seq Encoder-Decoder	27
7.1 Methodology	27
7.2 Data Preparation	27
7.3 Modeling	29
VIII. Conclusion	31
IX. Technical Deliverable	33
X. Reference & Credit	45

I. Project Definition & Business Objectives

1.1 What Is the Business Problem?

Sales forecasting is essential for running a successful business, especially for businesses in the retail industry. Accurately predicting unit product sales at different points of time is crucial for retail stores in meeting consumer demand and managing product inventory; however, it can be a complex problem with many factors such as products coming from various of categories, different store location may be running different promotion and special events, change of pricing, etc. This forecasting problem is further complicated by its hierarchical time series sales data.

Walmart Inc. is an American multinational retail corporation headquartered in the United States. It operates a chain of retail stores and e-commerce stores in 24 countries. For this business problem, we are making point forecasts of unit sales of ten Walmart stores' retail goods across three states California, Texas, and Wisconsin for two 28 day time periods.

1.2 What Is the Objective of this Project?

In this project, our goal is to predict the unit sales of 3049 items from seven departments and three product categories for ten stores across three states. The accuracy of our predictions will be measured automatically by the Kaggle platform by comparing our predictions to the actual sales of the two 28 day time periods in their datasets.

From the business perspectives, successfully predicting the daily sales would allow the organization to implement this algorithm for all stores across the globe to improve their operational efficiency in inventory planning and better serve their customers. In addition, knowing actual demand of products also reveals opportunities in product sourcing and making better decisions in marketing strategy.

II. Technical Specifications & Solution Overview

2.1 Tools

- Pandas
- Numpy
- Sklearn
- Lightgbm
- Scipy
- Pickle

2.2 Datasets

The datasets we used in this project were provided by the Kaggle competition. There are four main data files, and below are the descriptions of each files:

1. calendar.csv - This file contains information about the dates on which the products are sold.
2. sales_train_validation.csv - This file contains the historical daily unit sales data per product and store from day 1 to day 1919.
3. sell_prices.csv - This file contains information about the price of the products sold per store and date.
4. sales_train_evaluation.csv - This file contains the same historical daily unit sales data per product and store like the sales_train_validation dataset, but it also contains sales data from day 1920 to day 1941.

2.3 Solution Overview

Our solutions aims to solve this forecasting problem in three different ways:

1. Ensembling Model-Boosting

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages:¹

- Faster training speed and higher efficiency.
- Lower memory usage.
- Better accuracy.
- Support of parallel, distributed, and GPU learning.

¹resource: <https://lightgbm.readthedocs.io/en/v3.3.2/>

- Capable of handling large-scale data.

2. Time Series Statistical Models

SARIMAX- “autoregressive integrated moving average.” It’s a model used in statistics and econometrics to measure events that happen over a period of time. The model is used to understand past data or predict future data in a series. It’s used when a metric is recorded in regular intervals, from fractions of a second to daily, weekly or monthly periods.²

Prophet: is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.³

3. Recurrent Neural Networks

Seq to Seq: is a method of encoder-decoder based machine translation and language processing that maps an input of sequence to an output of sequence with a tag and attention value. The idea is to use two RNNs that will work together with a special token and try to predict the next state sequence from the previous sequence.⁴

Temporal Fusion Transformers: Google’s Temporal Fusion Transformer is an attention-based deep neural network. It mimics how the human brain can focus on certain bits of information while filtering out others.⁵

The detailed implementation of three approaches will be introduced later in this report, including data preparation, feature engineering, and model tuning. The best model will be chosen based on performance and running time due to resource constraints. The findings and recommendations will be provided based on the best model’s features importances.

²:<https://www.mastersindatascience.org/learning/statistics-data-science/what-is-arima-modeling/>

³ <https://github.com/facebook/prophet>

⁴ <https://www.guru99.com/seq2seq-model.html>

⁵<https://opendatascience.com/google-ai-proposes-temporal-fusion-transformer-for-multi-horizon-time-series-forecasting>

III. Predictive Model 1 - Ensembling Model with LightGBM

3.1 Methodology

To perform the Ensembling Model, we firstly need to transform the data to have each day's sales for each item as a record to store the target variable. The next step is to join calendar and sale prices information to add daily features to predictive factors. Feature engineering is also implemented to help improve model performance. Finally we adopted a hierarchical training framework to produce multiple models that balance data relevance and volume for a combined prediction.

3.2 Data preparation

1. Sales Train

Since the goal is to predict every day's unit sales of each item, we first reshaped the data into daily levels, in which each row represented a sales record of a specific date. After resharing the data, we also created our label column 'demand' which is the sales quantity value.

```
df=pd.melt(x,id_vars=['id','item_id','dept_id','cat_id','store_id','state_id'],va  
r_name='d',value_name='demand')
```

2. Calendar Info

This dataset allows us to identify whether a specific date is a weekday or weekend and if it is having special events or holidays or running the Supplemental Nutrition Assistance Program. Even though the event columns show a lot of missing values, these missing values actually indicates that there is no event on that day, thus we replaced them with "No event".

```
cat=['event_name_1','event_type_1','event_name_2','event_type_2']  
for i in cat:  
df[i].fillna('no_event',inplace=True)
```

3. Price

This dataset provides useful price information for each item at different store locations at different points of time, and the price is updated weekly. However, we noticed that an item has no sales record in a particular week, the price of that product would not be updated, and thus the price info

of that week would be missing. We handled these missing price values by imputing the average price of that item after integration.

4. Integration

We merged the sales train data with the Calendar and Price data frame to get our final data frame, so each daily sales record also contains its calendar and price information.

```
df=pd.merge(df,calendar_,on='d',how='left')
df=pd.merge(df,sell_prices_,on=['item_id','store_id','wm_yr_wk'],how='left')
df['sell_price']=df.groupby(['id'])['sell_price'].apply(lambda x:
x.fillna(x.mean()))
```

3.3 Feature engineering⁶

1. Categorical Features

We used label encoder to transform all the categorical features

```
labelencoder=LabelEncoder()

category=['event_name_1','event_type_1','event_name_2','event_type_2','id','item_
id','dept_id','cat_id','store_id','state_id']
for i in category:
    df[i+'_']=labelencoder.fit_transform(df[i])

df=df.drop(['event_name_1','event_type_1','event_name_2','event_type_2','id','ite
m_id','dept_id','cat_id','store_id','state_id'],axis=1)
```

2. Lag Features

In order to use past records as reference for future prediction, weekly lags features are created on item sale, sell price, store level average sale and store+department level average. 28, 35 and 42 are used as the lag basis. The rolling mean of lag features are also included.

```
# Get sales by 3 aggregation level: store_id, item_id, store_id+dept_id
store_demand =
df.groupby(['store_id','d'])['demand'].sum().reset_index().rename(columns={'deman
d':'store_demand'})
df = pd.merge(df, store_demand, on=['store_id','d'],how='left')
del store_demand
```

⁶ <https://dipanshurana.medium.com/m5-forecasting-accuracy-1b5a10218fcf>


```

item_demand =
df.groupby(['item_id', 'd'])['demand'].sum().reset_index().rename(columns={'demand': 'item_demand'})
df = pd.merge(df, item_demand, on=['item_id', 'd'], how='left')
del item_demand

store_dept_demand =
df.groupby(['store_id', 'dept_id', 'd'])['demand'].sum().reset_index().rename(columns={'demand': 'store_dept_demand'})
df = pd.merge(df, store_dept_demand, on=['store_id', 'dept_id', 'd'], how='left')
del store_dept_demand

# Create lag variables
lag_col_prefixs =
['lag_', 'lag_price_', 'lag_store_demand_', 'lag_item_demand_', 'lag_store_dept_demand_']

lag_col_names =
['demand', 'sell_price', 'store_demand', 'item_demand', 'store_dept_demand']
for col_prefix, col_name in zip(lag_col_prefixs, lag_col_names):
    lags=[28, 35, 42]
    lag_cols=[]
    for i in lags:
        df[col_prefix+str(i)]=df.groupby(['id'])[col_name].shift(i)
        lag_cols.append(col_prefix+str(i))
    # lag_cols=['lag_28', 'lag_35', 'lag_42']
    for i in lag_cols:
        df[i].fillna(0, inplace=True)

for val in [7, 30, 60, 90, 180]:
    df[f"rolling_std_t{val}"] =
df.groupby(["id"])["demand"].transform(lambda x: x.shift(28).rolling(val).std())
for val in [7, 30, 60, 90, 180]:
    df[f"rolling_mean_t{val}"] = df.groupby(["id"])["demand"].transform(lambda
x: x.shift(28).rolling(val).mean())

df["rolling_skew_t30"] = df.groupby(["id"])["demand"].transform(lambda x:
x.shift(28).rolling(30).skew())
df["rolling_kurt_t30"] = df.groupby(["id"])["demand"].transform(lambda x:
x.shift(28).rolling(30).kurt())

```

3. Time Features

Since this is a time series forecasting problem, we are interested in different time features. We believe knowing whether a day is a holiday or is having a special event would be helpful, as consumer's shopping patterns tend to change around these specific days. From the data, we learned that there are four different event types, which are sporting, cultural, national, religious. We also created a feature to indicate whether a day is a holiday. Thus, we created events time features as below:

```
# IsSporting, IsCulture, IsNational, IsReligious, and IsHoliday
df["IsSporting"] = np.where((df["event_type_1"] == "Sporting") |
(df["event_type_2"] == "Sporting"), 1, 0)
df["IsCultural"] = np.where((df["event_type_1"] == "Cultural") |
(df["event_type_2"] == "Cultural"), 1, 0)
df["IsNational"] = np.where((df["event_type_1"] == "National") |
(df["event_type_2"] == "National"), 1, 0)
df["IsReligious"] = np.where((df["event_type_1"] == "Religious") |
(df["event_type_2"] == "Religious"), 1, 0)
df["IsHoliday"] = np.where((df["IsSporting"] == 1) | (df["IsCultural"] == 1) |
(df["IsNational"] == 1) | (df["IsReligious"] == 1) , 1, 0)
```

Besides using the time column in the original dataset, we also think knowing whether a day is a weekend, the day of the month, and the week of the month would be helpful and interesting because during these special circumstances, consumers' shopping behaviors may change. For instance, people may be more likely to shop during weekends as they have more time or need to get groceries for the upcoming week, thus time features below have been created.

- is_weekend, month_day, month_week

```
# Create is_weekend feature
f=lambda x: 1 if x<=2 else 0
df['is_weekend']=df['wday'].map(f)
df['is_weekend']=df['is_weekend'].astype(np.int8)

# Create month_day feature
f=lambda x: x.split("-")[2]
df['month_day']=df['date'].map(f)
df['month_day']=df['month_day'].astype(np.int8)

# Create month_week_number feature
df['month_week_number']=(df['month_day']-1) // 7 + 1
df['month_week_number']=df['month_week_number'].astype(np.int8)
```

4. Price features

Price is the most important factor for customer demands, hence we did major feature engineering based on price features, including:

- lag 1 day price and recent price change
- price changes in the past 1 year to measure an item's price variance

```
# price features
df['lag_price_t1'] = df.groupby(['id'])['sell_price'].transform(lambda x:
x.shift(1))
df['price_change_t1'] = (df['lag_price_t1'] - df['sell_price']) /
(df['lag_price_t1'])
df['rolling_price_max_t365'] = df.groupby(['id'])['sell_price'].transform(lambda
x: x.shift(1).rolling(365).max())
df['price_change_t365'] = (df['rolling_price_max_t365'] - df['sell_price']) /
(df['rolling_price_max_t365'])
df['rolling_price_std_t7'] = df.groupby(['id'])['sell_price'].transform(lambda
x: x.rolling(7).std())
df['rolling_price_std_t30'] = df.groupby(['id'])['sell_price'].transform(lambda
x: x.rolling(30).std())
df.drop(['rolling_price_max_t365', 'lag_price_t1'], inplace = True, axis = 1)
```

3.4 Algorithm Selection

Given the scale of the input dataset, it takes hours to train and tune one model, in the interest of time we chose LightGBM over XGBoost as the single boosting method.

3.5 Parameter Tuning

When choosing the training objective, we used “tweedie” instead of the default “regression”. The distribution of our target variable “sales” is highly skewed towards zero, rather than conforming well to the normal distribution, therefore “tweedie” works better⁷.

```
lgb = LGBMRegressor(objective="tweedie", tweedie_variance_power=1.1)
```

For the rest of hyperparameter tuning, we used a 5-fold Cross-Validation splits to compare the performance (as mean squared error). To avoid data leakage, we used a custom split function to manually pull out the last 5 28-day data as validation set and all the other data ahead each split as training set respectively.

⁷ What is Tweedie: <https://medium.com/@royraavid/have-you-heard-about-tweedie-loss-bb94551dd82f>

```
def custom_split(X, y, groups):
    for train_index, test_index in groups:
        original_train_index = np.array(X[X['day'].isin(train_index+1)].index)
        original_test_index = np.array(X[X['day'].isin(test_index+1)].index)
        yield original_train_index, original_test_index

tscv = TimeSeriesSplit(n_splits=5, test_size=28)
groups = tscv.split(df1['day'].unique())
```

Due to the large volume of data and the number of models we need to train, it is very time-consuming to use GridSearch for hyperparameters tuning, instead we used RandomizedSearch and with a limited search space consisted of key hyperparameters.

```
lgb_grid = {'learning_rate': reciprocal(1e-3, 3e-1),
            'max_depth': list(range(3, 12)) ,
            'num_leaves': list(range(20, 3000, 20)),
            'n_estimators': sp.stats.randint(low=400, high=1500),
            "min_gain_to_split": list(range(0, 15)),
            'max_bin': sp.stats.randint(low=200, high=300),
            "min_data_in_leaf": list(range(200, 3000, 100)),
            "lambda_l1": list(range(0, 100, 5)),
            "lambda_l1": list(range(0, 100, 5)),
            "min_gain_to_split": list(range(0, 15)),
            "feature_fraction": sp.stats.uniform(loc=0.2, scale=0.7)
            }

lgb_reg = RandomizedSearchCV(lgb, param_distributions=lgb_grid, n_jobs=-1,
                             scoring = 'neg_mean_squared_error',
                             cv=custom_split(X, y, groups))
```

The best parameters vary between different models, see appendix for details.

3.6 Training Strategy⁸

Based on this data we will predict sales of 3049 items for 10 stores from CA, TX and WI. These items are from 7 departments that belong to Hobbies, Household and Foods. Under this hierarchical structure, it is natural for us to train and predict at different levels.

We tried a training model by store and by store - department to get 2 sets of predictions, and also combined the 2 predictions with a simple average to produce a 3rd one.

⁸ <https://www.kaggle.com/c/m5-forecasting-accuracy/discussion/163216>
<https://www.kaggle.com/c/m5-forecasting-accuracy/discussion/164599>

Meanwhile since our prediction period is 4-week, we also tried to split the dataset into 4 splits based on the week number within each 4-week period, so for each week prediction we used $\frac{1}{4}$ of the training data. However it did not see a performance improvement.

Finally, considering different models may capture a certain part of the pattern, we tried to combine the predictions from models, specifically from the store and store-department level predictions, and the combined outcome produced the least error.

Find the model performance summary in below chart:

LGBM Model Approaches we tried	Kaggle Private Score
LGBM by Store	0.56183
LGBM by Store and by Department	0.575
LGBM by week	0.66
Combination of Approaches	0.5454

IV. Predictive Model 2 - SARIMAX

4.1 Methodology

Seasonal Auto-Regressive Integrated Moving Average with eXogenous factors, or SARIMAX, is an extension of the ARIMA class of models. Intuitively, SARIMAX models compose 5 parts:

- **S: Seasonality.** Seasonality is that in the time series data, there exist repeatable and predictable patterns.
- **AR: Autoregression.** A model that uses the relationship between an observation and some number of lagged observations
- **I: Integrated.** The use of differencing of raw observations to make the time series stationary.
- **MA: Moving Average.** A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.
- **X: Exogenous Variable.** The exogenous variable is used to model the remaining error, including information that is not recurrent over time.

SARIMA models are usually denoted as $(p,d,q)(P,D,Q)S$. Each of the parameters indicates the specific SARIMA model being used. (p,d,q) are non-seasonal orders, p is the autoregressive order, d is the differencing order and q is the moving average order. On the other hand, (P,D,Q) are seasonal orders, P is the seasonal autoregressive order, D is the seasonal differencing order, Q is the seasonal moving average order and finally S is the number of time steps per cycle.

$$\text{SARIMA } (p, d, q) \times (P, D, Q, S)$$

SARIMAX extends on this framework just by adding the capability to handle exogenous variables. Holidays is the go-to option.

In order to find the best parameters for the SARIMAX model, we use the AutoArima function from the package: pmdarima to automatically identify the best parameters.

4.2 Data preparation

4.2.1 Training data

To better represent the holiday, we extract the category of the holidays from both of the two event columns. The extracted data served as the exogenous variables in the model.

```
dates["IsSporting"] = np.where((dates["event_type_1"] == "Sporting") |
```

```

(dates["event_type_2"] == "Sporting"), 1, 0)
dates["IsCultural"] = np.where((dates["event_type_1"] == "Cultural") |
(dates["event_type_2"] == "Cultural"), 1, 0)
dates["IsNational"] = np.where((dates["event_type_1"] == "National") |
(dates["event_type_2"] == "National"), 1, 0)
dates["IsReligious"] = np.where((dates["event_type_1"] == "Religious") |
(dates["event_type_2"] == "Religious"), 1, 0)
dates["IsHoliday"] = np.where((dates["IsSporting"] == 1) | (dates["IsCultural"]
== 1) | (dates["IsNational"] == 1) | (dates["IsReligious"] == 1) , 1, 0)

```

4.2.2 Prediction data

We also need exogenous data for the prediction. In order to find out the fixed holiday, we check the dates from 06-20 to 07-17 in the previous years and only preserve those holidays that happened every year on the same day.

```

exo = exo.groupby("date").agg(snap_CA = ("snap_CA", np.mean),
                             snap_TX = ("snap_TX", np.mean),
                             snap_WI = ("snap_WI", np.mean),
                             IsSporting = ("IsSporting", np.mean),
                             IsCultural = ("IsCultural", np.mean),
                             IsNational = ("IsNational", np.mean),
                             IsReligious = ("IsReligious", np.mean),
                             IsHoliday = ("IsHoliday", np.mean)).reset_index()

```

```

exo = exo.groupby("date").agg(snap_CA = ("snap_CA", np.mean),
                             snap_TX = ("snap_TX", np.mean),
                             snap_WI = ("snap_WI", np.mean),
                             IsSporting = ("IsSporting", np.mean),
                             IsCultural = ("IsCultural", np.mean),
                             IsNational = ("IsNational", np.mean),
                             IsReligious = ("IsReligious", np.mean),
                             IsHoliday = ("IsHoliday", np.mean)).reset_index()

```

```

exo = exo.replace([0.2, 0.4], 0.0)

```

4.3 Modeling

We have to build a single model for each product in every store, namely 30490 models in total. For automation, we build a loop to build each model.

```
import pmdarima as pm
predictions = pd.DataFrame(columns=["store_id", "item_id", "date", "prediction"])

for i in stores:
    for j in items:
        d = df[(df["store_id"] == i) & (df["item_id"] == j)]
        d.drop(["store_id", "item_id", "d"], axis = "columns")

        test_days = 28
        train = d.iloc[:, :]
        test = exo
        # test = d.iloc[-test_days:, :]

        train_exog = train.iloc[:, 2:]
        # test_exog = test.iloc[:, 2:]
        test_exog = exo.iloc[:, 0:]

        model = pm.auto_arima(y = train['sales'],
                              exogenous = train_exog,
                              m = 7,
                              seasonal = True,
                              stepwise = True)

        predictions_sarimax = pd.Series(model.predict(n_periods= test_days,
                                                       X = test_exog)).rename("prediction")
        predictions_sarimax.index = test.index
        sarimax_df=predictions_sarimax.to_frame()
        sarimax_df=sarimax_df.reset_index()
        sarimax_df['store_id']=i
        sarimax_df['item_id']=j

        predictions.append(sarimax_df)
```


V. Predictive Model 3 - Prophet

5.1 Methodology

Prophet is an open-source project from Facebook that aims to model time series data and make reliable forecasts. Essentially, a Prophet model is a generalized additive model with several components: a seasonal component with multiple levels (e.g., yearly, weekly, daily, and user-defined seasonalities), a trend component which can be highly non-linear, and holiday effects.

Prophet models are probabilistic machine learning models, meaning that their predictions include uncertainty measures such as 95% credible intervals. Prophet fits models with Stan, a probabilistic programming language, and the default learning strategy is maximum a posteriori (MAP).

The model can typically be expressed with the following formula:

$$y_t = g(t) + s(t) + h(t)$$

where:

- t is the time index
- y_t is the time series of interest
- $g(t)$ is a (possibly smooth and nonlinear) function for the trend
- $s(t)$ represents the seasonal, periodic changes (e.g., yearly and weekly seasonalities)
- $h(t)$ models the effects of holidays, whose schedule may be irregular (e.g., Christmas, Easter)

Prophet is fully automatic with no manual effort. We can also add our own parameter space to improve the forecast.

5.2 Data preparation

5.2.1 Training data

To better represent the holiday, we extract the category of the holidays from both of the two event columns. The extracted data served as the exogenous variables in the model.

```
dates["IsSporting"] = np.where((dates["event_type_1"] == "Sporting") |  
                                (dates["event_type_2"] == "Sporting"), 1, 0)  
dates["IsCultural"] = np.where((dates["event_type_1"] == "Cultural") |  
                                (dates["event_type_2"] == "Cultural"), 1, 0)
```

```

dates["IsNational"] = np.where((dates["event_type_1"] == "National") |
                                (dates["event_type_2"] == "National"), 1, 0)
dates["IsReligious"] = np.where((dates["event_type_1"] == "Religious") |
                                (dates["event_type_2"] == "Religious"), 1, 0)
dates["IsHoliday"] = np.where((dates["IsSporting"] == 1) | (dates["IsCultural"]
== 1) | (dates["IsNational"] == 1) | (dates["IsReligious"] == 1) , 1, 0)

```

According to the Prophet, dates should be named as ds and the predicted variable should be named as y.

```

df = df.rename(columns = {'date' : 'ds', 'sales' : 'y'})

```

5.2.2 Prediction data

We also need exogenous data for the prediction. In order to find out the fixed holiday, we check the dates from 06-20 to 07-17 in the previous years and only preserve those holidays that happened every year on the same day.

```

exo = dates[["date", "snap_CA", "snap_TX", "snap_WI", "IsSporting", "IsCultural",
"IsNational", "IsReligious", "IsHoliday"]]
exo["date"] = exo["date"].str[5:10]
exo = exo[(exo["date"] >= "06-20") & (exo["date"] <= "07-17")]
exo['date'] = '2016-' + exo['date'].astype(str)

```

```

exo = dates[["date", "snap_CA", "snap_TX", "snap_WI", "IsSporting", "IsCultural",
"IsNational", "IsReligious", "IsHoliday"]]
exo["date"] = exo["date"].str[5:10]
exo = exo[(exo["date"] >= "06-20") & (exo["date"] <= "07-17")]
exo['date'] = '2016-' + exo['date'].astype(str)

```

The Prophet also requires the date column to be named as ds.

```

exo = exo.replace([0.2, 0.4], 0.0).rename(columns={"date": "ds"})

```

5.3 Modeling

We have to build a single model for each product in every store, namely 30490 models. Meanwhile, in order to find the best hyperparameter set, we provide a search space with:

- 'seasonality_prior_scale': [0.01, 0.1, 1.0] and,
- 'changepoint_prior_scale': [0.01, 0.05, 0.1]

For each model, we will try 9 hyperparameter sets to find out the one with best performance. Thus, we have trained 274410 models in total.

Here, we also build a loop to build the 274k models automatically.

```
from prophet import Prophet
from prophet.diagnostics import cross_validation, performance_metrics
stores = df["store_id"].unique()
items = df["item_id"].unique()

from sktime.performance_metrics.forecasting import MeanSquaredScaledError
import itertools
import numpy as np
import pandas as pd

param_grid = {'seasonality_prior_scale': [0.01, 0.1, 1.0],
              'changepoint_prior_scale': [0.01, 0.05, 0.1]}
all_params = [dict(zip(param_grid.keys(), v)) for v in
itertools.product(*param_grid.values())]

predictions = pd.DataFrame(columns=["store_id", "item_id", "date", "prediction"])

for i in stores:
    for j in items:
        d = df[(df["store_id"] == i) & (df["item_id"] == j)]
        d.drop(["store_id", "item_id", "d"], axis = "columns")

        train = d.iloc[:, :]

        RMSE = []

        for params in all_params:
            m = Prophet(growth = "linear",
                        yearly_seasonality = True,
                        weekly_seasonality = True,
                        daily_seasonality = False,
                        seasonality_mode = "multiplicative",
                        **params
                        )
            m.add_regressor("IsHoliday")
            m.add_regressor("IsSporting")
```

```

m.add_regressor("IsCultural")
m.add_regressor("IsNational")
m.add_regressor("IsReligious")
m.add_regressor("snap_CA")
m.add_regressor("snap_TX")
m.add_regressor("snap_WI")

m.fit(train)

df_cv = cross_validation(m,
                        period='28 days',
                        horizon = '28 days', # predict
                        initial = '84 days',
                        parallel = "processes")

df_p = performance_metrics(df_cv, rolling_window=1)
RMSE.append(df_p['rmse'].values[0])

best_params = all_params[np.argmin(RMSE)]

bm = Prophet(growth = "linear",
             yearly_seasonality = True,
             weekly_seasonality = True,
             daily_seasonality = False,
             seasonality_mode = "multiplicative",
             **best_params)
bm.add_regressor("IsHoliday")
bm.add_regressor("IsSporting")
bm.add_regressor("IsCultural")
bm.add_regressor("IsNational")
bm.add_regressor("IsReligious")
bm.add_regressor("snap_CA")
bm.add_regressor("snap_TX")
bm.add_regressor("snap_WI")

bm.fit(train)

# future = bm.make_future_dataframe(periods=28, freq='d')
future = exo
forecast = m.predict(future)

bm_df = forecast[['ds', 'yhat']].rename(columns = {'ds' : 'date', 'yhat' :
'prediction'})

bm_df['store_id']=i

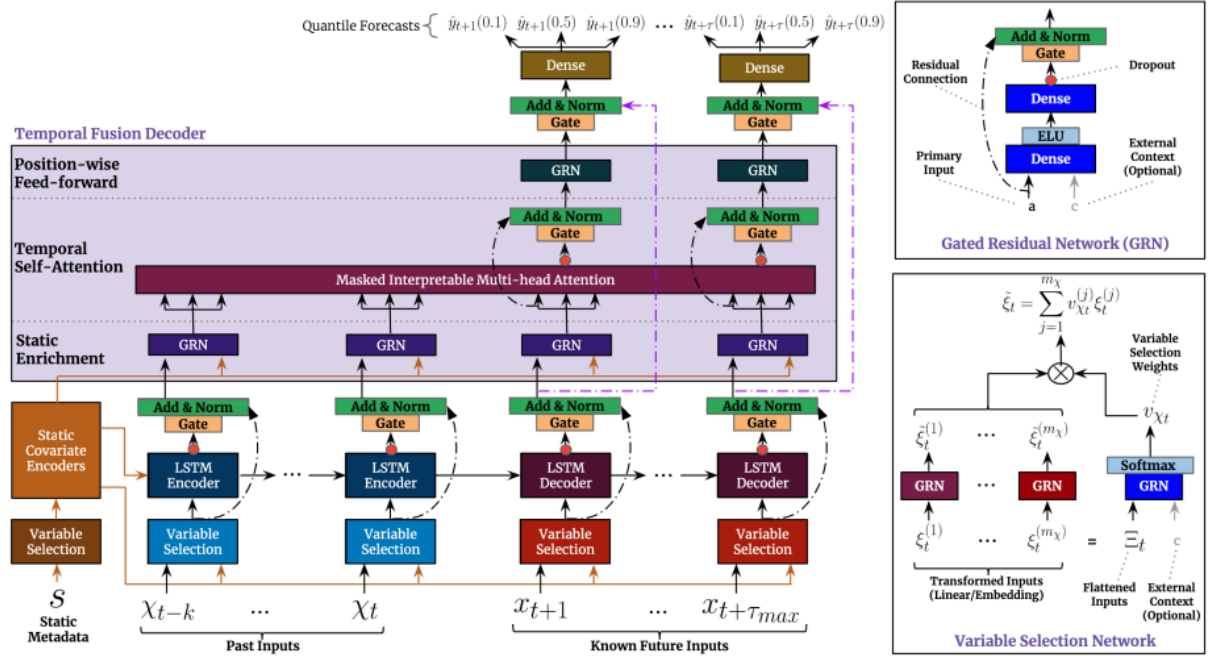
```

```
bm_df['item_id']=j  
  
predictions = predictions.append(bm_df)
```

VI. Predictive Model 4 - Temporal Fusion Transformer

6.1 Methodology

Temporal Fusion Transformer (TFT) is an attention-based time-series technique that offers high-performance multi-horizon forecasting and also can provide interpretable insights. TFT has 5 major components which can solve wide range of problems that output high performance. The network outperforms DeepAR by Amazon by 36-69% in benchmarks.



1. **Gating mechanisms** to skip over any unused components of the architecture, providing adaptive depth and network complexity to accommodate a wide range of datasets and scenarios.
2. **Variable selection networks** to select relevant input variables at each time step.
3. **Static covariate encoders** to integrate static features into the network, through encoding of context vectors to condition temporal dynamics.
4. **Temporal processing** to learn both long- and short-term temporal relationships from both observed and known time-varying inputs. A sequence to sequence layer is employed for local processing, whereas long-term dependencies are captured using a novel interpretable multi-head attention block.
5. **Prediction intervals** via quantile forecasts to determine the range of likely target values at each prediction horizon.

6.2 Data Preparation

We used Pytorch to implement the Temporal Fusion Transformer. New features were added to improve the accuracy of the prediction. Since the TFT model has the ability to predict multi-horizon time series problem, we added several sales volume columns at different level of aggregation.

```
df_subset["avg_volume_by_state"] = df_subset.groupby(["day", "state_id"],
observed=True).sales.transform("mean")
df_subset["avg_volume_by_store"] = df_subset.groupby(["day", "store_id"],
observed=True).sales.transform("mean")
df_subset["avg_volume_by_product"] = df_subset.groupby(["day", "item_id"],
observed=True).sales.transform("mean")
df_subset["avg_volume_by_dept"] = df_subset.groupby(["day", "dept_id"],
observed=True).sales.transform("mean")
df_subset["log_num_sold"] = np.log(df_subset.sales + 1e-8)
df_subset["avg_volume_by_category"] = df_subset.groupby(["day", "cat_id"],
observed=True).sales.transform("mean")
gc.collect()
```

Pytorch uses a special format of dataset. Thus, the dataset has to be transformed using TimeseriesDataset function. In our case, we are using the first 1913 days of sale as our input, and predict the last 28 days of item sales. The TimeSeriesDataset function provided by pytorch integrated lots of data transformation functions including normalization, categorical encoder and add lag features. For the lag columns, we used 7,14, and 28 days.

```
train= df_subset[df_subset['day']<=1941]
test=df_subset[df_subset['day']>1941]

max_prediction_length = 28
max_encoder_length = 72
training_cutoff = train["day"].max() - max_prediction_length

# Let's create a Dataset
training = TimeSeriesDataSet(
    train[lamba x: x.day <= training_cutoff],
    time_idx="day",
    target="sales",
    group_ids=[ "store_id", "dept_id", "item_id"],
    min_encoder_length=max_prediction_length//2, # keep encoder length Long (as
it is in the validation set)
    max_encoder_length=max_encoder_length,
```

```

min_prediction_length=1,
max_prediction_length=max_prediction_length,
static_categoricals=[ 'state_id','dept_id',"store_id",
                      "item_id","snap_CA","snap_TX","snap_WI"],
time_varying_known_categoricals=[ 'wm_yr_wk', 'wday', 'month'
                                  ],
#variable_groups={"is_holiday": ["is_holiday"]}, # group of categorical
variables can be treated as one variable
time_varying_known_reals=["day","sell_price"],
time_varying_unknown_categoricals=[],
time_varying_unknown_reals=[
    "sales", "log_num_sold", "avg_volume_by_state",
    "avg_volume_by_store", "avg_volume_by_category", "avg_volume_by_dept",
    "avg_volume_by_product"
],
target_normalizer=GroupNormalizer(
    groups=[ "store_id", "dept_id","item_id"], transformation="softplus"
), # use softplus and normalize by group

lags={'sales': [7, 14, 28]},
add_encoder_length=True,
add_relative_time_idx=True,
add_target_scales=True,
categorical_encoders={
    'item_id':NaNLabelEncoder(add_nan=True),
    'wm_yr_wk':NaNLabelEncoder(add_nan=True),
    'wday' : NaNLabelEncoder(add_nan=True),
    'month':NaNLabelEncoder(add_nan=True)
}

)

# create validation set (predict=True) which means to predict the last
max_prediction_length points in time
# for each series
validation = TimeSeriesDataSet.from_dataset(training, train, predict=True,
stop_randomization=True)

# create dataloaders for model
batch_size = 128 # set this between 32 to 128
train_dataloader = training.to_dataloader(train=True, batch_size=batch_size,
num_workers=0)
val_dataloader = validation.to_dataloader(train=False, batch_size=batch_size *
10, num_workers=0)

```


6.3 Modeling

Next, we train the model using pytorch lightning package. We use start with learning rate at 0.03 and max epoch at 120. We will also implement early stopping. For the TFT model itself, we will use 2 LSTM layers and 2 attention head. One advantage of the TFT model is that the training speed of the TFT model are not affected by the hidden layers. In our case, we picked 16 hidden layers. From the output we can see that, there are 69.2K parameters that need to be trained for the TFT model. The loss function here we used Tweedie deviance score which is an alternative to tweedie loss since tweedie loss is not implemented in the pytorch forecasting package.

```
PATIENCE = 30
MAX_EPOCHS = 120
LEARNING_RATE = 0.03
OPTUNA = False

early_stop_callback = EarlyStopping(monitor="train_loss", min_delta=1e-2,
patience=PATIENCE, verbose=False, mode="min")
lr_logger = LearningRateMonitor() # log the learning rate
logger = TensorBoardLogger("lightning_logs") # logging results to a tensorboard
trainer = pl.Trainer(
    max_epochs=MAX_EPOCHS,
    gpus=0,
    enable_model_summary=True,
    gradient_clip_val=0.25,
    limit_train_batches=10, # comment in for training, running validation every 30
batches
    #fast_dev_run=True, # comment in to check that network or dataset has no
serious bugs
    callbacks=[lr_logger, early_stop_callback],
    logger=logger,
)
tft = TemporalFusionTransformer.from_dataset(
    training,
    learning_rate=LEARNING_RATE,
    lstm_layers=2,
    hidden_size=16,
    attention_head_size=2,
    dropout=0.2,
    hidden_continuous_size=8,
    output_size=1, # 7 quantiles by default
    loss=TweedieDevianceScore(),
    log_interval=10, # uncomment for learning rate finder and otherwise, e.g. to
10 for logging every 10 batches
    reduce_on_plateau_patience=4
```

```
)
```

```
tft.to(DEVICE)
```

```
print(f"Number of parameters in network: {tft.size()/1e3:.1f}k")
```

	Name	Type	Params
0	loss	RMSE	0
1	logging_metrics	ModuleList	0
2	input_embeddings	MultiEmbedding	34.6 K
3	prescalers	ModuleDict	256
4	static_variable_selection	VariableSelectionNetwork	3.1 K
5	encoder_variable_selection	VariableSelectionNetwork	10.3 K
6	decoder_variable_selection	VariableSelectionNetwork	3.0 K
7	static_context_variable_selection	GatedResidualNetwork	1.1 K
8	static_context_initial_hidden_lstm	GatedResidualNetwork	1.1 K
9	static_context_initial_cell_lstm	GatedResidualNetwork	1.1 K
10	static_context_enrichment	GatedResidualNetwork	1.1 K
11	lstm_encoder	LSTM	4.4 K
12	lstm_decoder	LSTM	4.4 K
13	post_lstm_gate_encoder	GatedLinearUnit	544
14	post_lstm_add_norm_encoder	AddNorm	32
15	static_enrichment	GatedResidualNetwork	1.4 K
16	multihead_attn	InterpretableMultiHeadAttention	808
17	post_attn_gate_norm	GateAddNorm	576
18	pos_wise_ff	GatedResidualNetwork	1.1 K
19	pre_output_gate_norm	GateAddNorm	576
20	output_layer	Linear	17
69.2 K	Trainable params		
0	Non-trainable params		
69.2 K	Total params		
0.277	Total estimated model params size (MB)		

6.4 Hyperparameter Tuning

We used OPTUNA to tune our hyperparameter, which is built-in within the pytorch forecasting package. We explored different hyperparameter settings such as gradient clip val range from 0.01 to 1.0, hidden size range from 8 to 128 and learning rate from 0.001 to 0.1 etc.

```
trainer.fit(  
    tft,  
    train_dataloaders=train_dataloader,  
    val_dataloaders=val_dataloader,  
)
```

```
if OPTUNA:
```

```
    from pytorch_forecasting.models.temporal_fusion_transformer.tuning import  
    optimize_hyperparameters
```

```

# create study
study = optimize_hyperparameters(
    train_dataloader,
    val_dataloader,
    model_path="optuna_test",
    n_trials=50,
    max_epochs=50,
    gradient_clip_val_range=(0.01, 1.0),
    hidden_size_range=(8, 128),
    hidden_continuous_size_range=(8, 128),
    attention_head_size_range=(1, 4),
    learning_rate_range=(0.001, 0.1),
    dropout_range=(0.1, 0.3),
    trainer_kwargs=dict(limit_train_batches=30),
    reduce_on_plateau_patience=4,
    use_learning_rate_finder=False, # use Optuna to find ideal learning rate
or use in-built learning rate finder
)

best_model_path = trainer.checkpoint_callback.best_model_path
best_tft = TemporalFusionTransformer.load_from_checkpoint(best_model_path)
actuals = torch.cat([y[0] for x, y in iter(val_dataloader)])
predictions = best_tft.predict(val_dataloader, mode="prediction")
raw_predictions, x = best_tft.predict(val_dataloader, mode="raw", return_x=True)

sm = RMSE()
print(f"Validation median RMSE loss: {sm.loss(actuals, predictions).mean(axis =
1).median().item()}")

filename = 'final_TFT_model.sav'
pickle.dump(best_tft, open(filename, 'wb'))

```

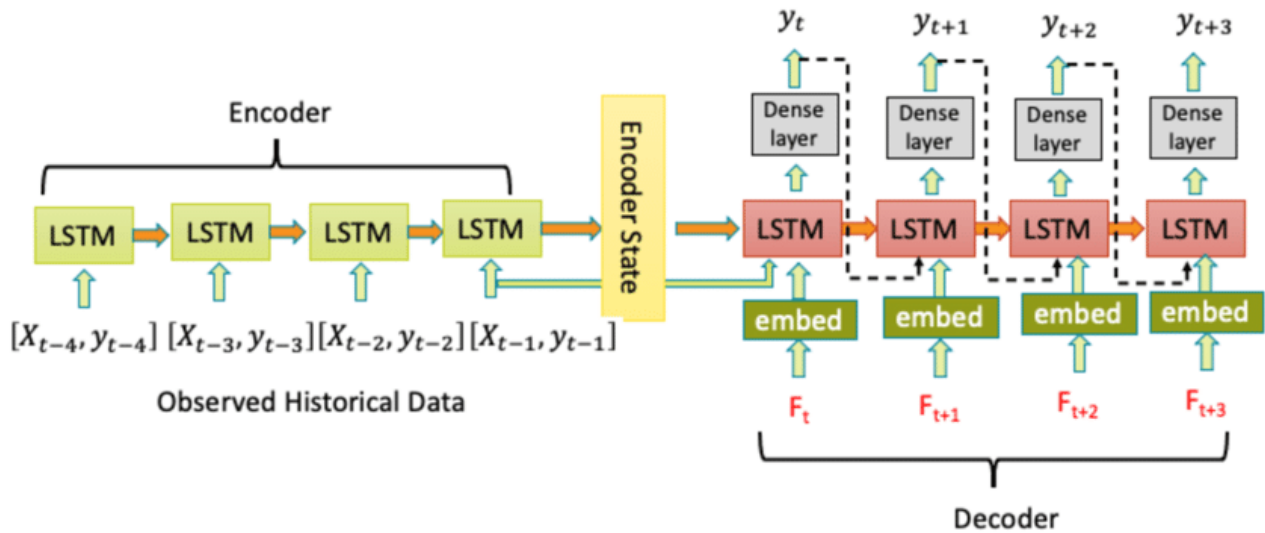
VII. Predictive Model 5 - Seq2Seq Encoder-Decoder

7.1 Methodology

Sequence to Sequence (often abbreviated to seq2seq) models is a special class of Recurrent Neural Network with encoder-decoder architecture used in sequence prediction tasks.⁹

Encoder: Encoder reads the input sequence and summarizes the information in something called the internal state vectors or context vectors (in case of LSTM these are called the hidden state and cell state vectors). We discard the outputs of the encoder and only preserve the internal states. This context vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions.

Decoder: The decoder is an LSTM whose initial states are initialized to the final states of the Encoder LSTM, i.e. the context vector of the encoder's final cell is input to the first cell of the decoder network. Using these initial states, the decoder starts generating the output sequence, and these outputs are also taken into consideration for future outputs. Each recurrent unit accepts a hidden state from the previous unit and produces an output as well as its own hidden state.



7.2 Data Preparation

1. Timestamps Related Features

⁹ <https://www.analyticsvidhya.com/blog/2020/08/a-simple-introduction-to-sequence-to-sequence-models/>

1. Time Features

Many of these features are inherently cyclical in nature, like the day of the week and month of year, in order to provide this information to the model, sine and cosine transformations were applied to the datetime features.

```
def sin_transform(values): return np.sin(2*np.pi*values/len(set(values)))
def cos_transform(values): return np.cos(2*np.pi*values/len(set(values)))
```

2. Boolean Features

Boolean features are created to keep track of special meanings of each selling day, including the start/end of year and month, weekend or weekday, special occasion, national holiday... etc.

In our Walmart case, binary columns were generated to indicate if the day is on a holiday, because we believed that holiday will attract more people to make purchases. Another boolean feature included in the original dataset is the Snap day.

2. Time Series Related Features

In time series forecasting, not only the data for current time but also data for certain previous time periods can also be added as the new features for prediction to help us to discover more trending factors and even some abnormal change. In this case, both lag features and rolling window statistics features are used.

1. Lag Features

Lag features simply means for data in current time t , when we push the time back to i steps and get data in $t-i$, this data will be highly correlated with current one and can be used for future prediction. In this dataset, we picked a time range from a week to 28 days after a week with a 7 days interval to create our lag features. The reason is we want the testing data(last 28 days) to have non-zero lag features as well.

```
# Lag
SHIFT_DAY = 7
TARGET = 'amt'
LAG_DAYS = [col for col in range(SHIFT_DAY,SHIFT_DAY+28,7)]
grid_df_sub = grid_df_sub.assign(**{
    '{}_lag_{}'.format(col, l):
    grid_df_sub.groupby(['id'])[col].transform(lambda x:x.shift(l))
    for l in LAG_DAYS
    for col in [TARGET]})
```

2. Rolling Window Statistics

Rolling features means we aggregate the previous i timesteps data with statistical methods. In our dataset, we chose mean and standard deviation these two methods. We picked two types of rolling

windows, one was directly rolling the previous 7,14 and 28 days data, another kind was to use three time windows as sliding shifts and rolling the previous 7, 14 and 28 days.

```
# Rollings
for i in [7,14,28]:
    grid_df_sub['rolling_mean_'+str(i)] =
        grid_df_sub.groupby(['id'])[TARGET]
        .transform(lambda x: x.shift(SHIFT_DAY).rolling(i).mean())

    grid_df_sub['rolling_std_'+str(i)] =
        grid_df_sub.groupby(['id'])[TARGET]
        .transform(lambda x: x.shift(SHIFT_DAY).rolling(i).std())

# Rollings with sliding shift
for d_shift in [1,7,14]:
    for d_window in [7,14,28]:
        col_name = 'rolling_mean_tmp_'+str(d_shift)+'_'+str(d_window)
        grid_df_sub[col_name] = grid_df_sub.groupby(['id'])[TARGET]
        .transform(lambda x: x.shift(d_shift)
        .rolling(d_window).mean()).astype(np.float16)
```

3. Variables Related Features

We applied one hot encoding on two categorical columns, item category and event type.

4. Scaling & Format Transformation & Split Data

We used a MinmaxScaler with a set range between -1 and 1 to apply data normalization.

One special thing is Seq2Seq model expects input data format as (num of sample, timesteps, num of features). For the Walmart case, we were using 28 days as both input and output timesteps. First step for preparation of input data was to split all data with a ratio between training, validation and testing data 6:3:1 respectively. Then a separate generator function was wrote to randomly create times series of 28 consecutive days, as results theses time series had following three dimensional shape: 30490 items * 28 days * num_features and ready to feed into the model.

7.3 Modeling

1. Model Architecture

We built the model with LSTM as the encoder and the decoder layer, both input layers would be fed with a series of 28 days and 32 features, and dropout is used to prevent overfitting. The final

output layer would be wrapped by a TimeDistributed layer and achieve multi-step forecasting by producing 28 output for each time step in sequence.¹⁰

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
encoder_input (InputLayer)	[(None, 28, 32)]	0	[]
decoder_input (InputLayer)	[(None, 28, 32)]	0	[]
encoder_lstm (LSTM)	[(None, 28, 256), (None, 256), (None, 256)]	295936	['encoder_input[0][0]']
decoder_lstm (LSTM)	[(None, 28, 256), (None, 256), (None, 256)]	295936	['decoder_input[0][0]', 'encoder_lstm[0][1]', 'encoder_lstm[0][2]']
time_distributed_1 (TimeDistributed)	(None, 28, 1)	257	['decoder_lstm[0][0]']
=====			
Total params: 592,129			
Trainable params: 592,129			
Non-trainable params: 0			

2. Training


The computation cost was very high(training one epoch for 3049 items needs three hours), so we decided to move on to other more efficient models and not further develop on this one.


¹⁰ <https://www.kaggle.com/code/ievgenvp/lstm-encoder-decoder-via-keras-lb-0-5/script>


VIII. Conclusion

Overview:

In order to accurately predict Walmart's ten store's sales across three states in the next 28 days, predictive machine learning methods were performed in three different approaches, ensembling mode, time series statistical, and recurrent neural network. After comparing the model performance and model runtime, LightGBM was chosen as the final predictive model because of the lowest RMSE and fastest running time. After fine tuning the LightGBM model using random search, we are able to achieve a WRMSSE of 0.5454.


 Search



 Featured Prediction Competition

M5 Forecasting - Accuracy

Estimate the unit sales of Walmart retail goods

 University of Nicosia · 5,558 teams · 2 years ago

\$50,000

Prize Money

Overview

Data

Code

Discussion

Leaderboard

Rules

Team

Submissions

Late Submission

...

Submissions

You selected 0 of 1 submissions to be evaluated for your final leaderboard score. Since you selected less than 1 submission, Kaggle auto-selected up to 1 submissions from among your public best-scoring unselected submissions for evaluation. The evaluated submission with the best Private Score is used for your final score.

0/1

☒ Submissions evaluated for final score

All

Successful

Selected

Errors

Private Score ▾

Submission and Description

Private Score ⓘ

Public Score ⓘ

Selected



MSBA6421_Sec001_Simin_Rio_Lan_Yiyu_Sherry.csv

Complete (after deadline) · 10m ago

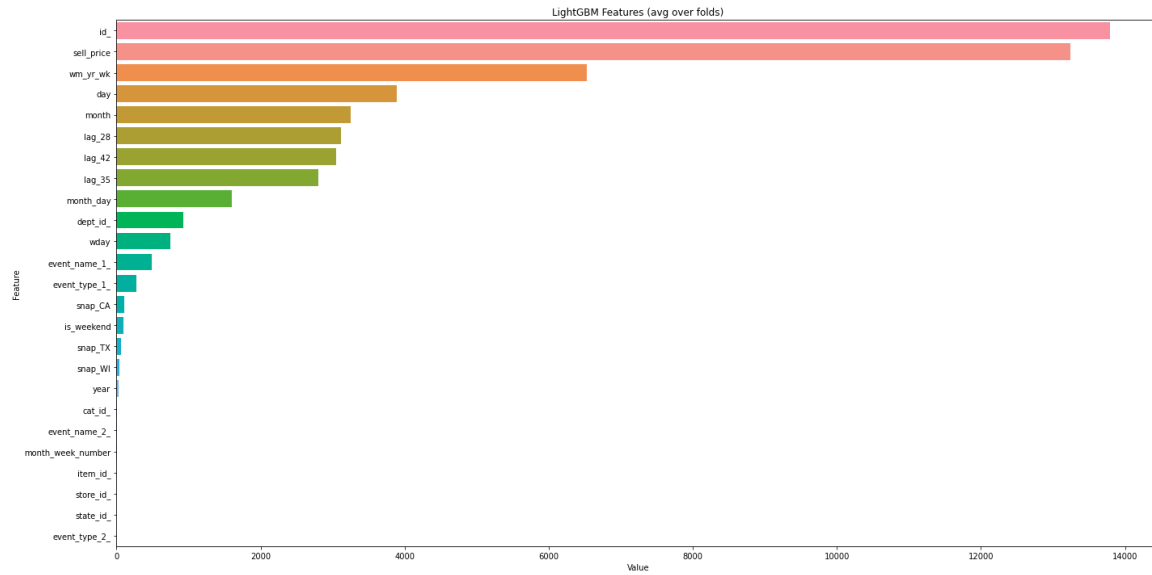
0.5454

5.44561



Findings:

Based on the features importance from our LightGBM models, we can see that id ranks at the top and follows by sell_price and wm_yr_wk. So beside the obviously features that sell prices will affect the sales amount significantly, we also found out that Walmart's customer in the three states purchase habit are consistent, most of the important features are related to timeseries. Also, special events for the day also influence the sales amount as expected.



Future work and implementation:

Due to resources and time constraints, more features could be added to the LightGBM model such as holidays/events lag. RNN models variants such as Prophet and Temporal Fusion Transformers could also perform well if times permits for fine tuning. The trained LightGBM model could be used for assisting inventory management and staff management in each store

IX. Technical Deliverable

Lightgbm

```
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

In [37]:

```
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error
from lightgbm import LGBMRegressor
import pickle
from sklearn.metrics import make_scorer
from sklearn.model_selection import TimeSeriesSplit
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import mean_squared_error
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))
```

In [11]:

```
# from google.colab import drive

# drive.mount('/content/drive')

# %cd '/content/drive/Shared drives/Predictive Analysis- Walmart'
```

In [12]:

```
sales_train_evaluation_ = pd.read_csv('/content/sales_train_evaluation.csv')
calendar_ = pd.read_csv('/content/calendar.csv')
sell_prices_ = pd.read_csv('/content/sell_prices.csv')
STORES_IDS = list(sales_train_evaluation_['store_id'].unique())
```

In [13]:

```
def reduce_mem_usage(df, verbose=True):
    numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
```

```

start_mem = df.memory_usage().sum() / 1024**2

for col in df.columns:

    col_type = df[col].dtypes

    if col_type in numerics:

        c_min = df[col].min()
        c_max = df[col].max()

        if str(col_type)[:3] == 'int':

            if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:

                df[col] = df[col].astype(np.int8)

            elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:

                df[col] = df[col].astype(np.int16)

            elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:

                df[col] = df[col].astype(np.int32)

            elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:

                df[col] = df[col].astype(np.int64)

        else:

            if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).max:

                df[col] = df[col].astype(np.float16)

            elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).max:

                df[col] = df[col].astype(np.float32)

            else:

                df[col] = df[col].astype(np.float64)

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

    if verbose: print('Mem. usage decreased to {:.2f} Mb ({:.1f}% reduction)'.format(end_mem, 100 * (start_mem -
end_mem) / start_mem))

return df

```

In [14]:

```

def data_processing(x):

    ##Adding columns for the days d_1942 to d_1969 as nan for which we need to forecast sales

    for i in range(1942,1970):

        x['d_'+str(i)]=np.nan

        x['d_'+str(i)]=x['d_'+str(i)].astype(np.float16)


    # Melting to convert one date to one observation

df=pd.melt(x,id_vars=['id','item_id','dept_id','cat_id','store_id','state_id'],var_name='d',value_name='demand')

```

```

# Merge calendar and sell_prices data
df=pd.merge(df,calendar_,on='d',how='left')

df=pd.merge(df,sell_prices_,on=['item_id','store_id','wm_yr_wk'],how='left')
df['sell_price']=df.groupby(['id'])['sell_price'].apply(lambda x: x.fillna(x.mean()))


# Fill events N/A
cat=['event_name_1','event_type_1','event_name_2','event_type_2']
for i in cat:
    df[i].fillna('no_event',inplace=True)


# Create is_weekend feature
f=lambda x: 1 if x<=2 else 0
df['is_weekend']=df['wday'].map(f)
df['is_weekend']=df['is_weekend'].astype(np.int8)


# Create month_day feature
f=lambda x: x.split("-")[2]
df['month_day']=df['date'].map(f)
df['month_day']=df['month_day'].astype(np.int8)


# Create month_week_number feature
df['month_week_number']=(df['month_day']-1) // 7 + 1
df['month_week_number']=df['month_week_number'].astype(np.int8)


# Lags
lags=[28,35,42]
for i in lags:
    df['lag_'+str(i)]=df.groupby(['id'])['demand'].shift(i)
df = df.fillna(0)

# # Rolling Median
# window=[7,14,28,35,42]
# for i in window:
#     df['rolling_median_'+str(i)]=df.groupby(['id'])['demand'].transform(lambda s: s.rolling(i,center=False).median())

```

```

# window=['rolling_median_7','rolling_median_14','rolling_median_28','rolling_median_35','rolling_median_42']

# for i in window:

#     df[i]=df[i].fillna(0)


# Encoding

labelencoder=LabelEncoder()

category=['event_name_1','event_type_1','event_name_2','event_type_2','id','item_id','dept_id','cat_id','store_id','state_id']

for i in category:

    df[i+'_']=labelencoder.fit_transform(df[i])


df=df.drop(['event_name_1','event_type_1','event_name_2','event_type_2','id','item_id','dept_id','cat_id','store_id','state_id'],axis=1)


# Convert string date to int date

f=lambda x: x.split('_')[1]

df['day']=df['d'].map(f)

df['day']=df['day'].astype(np.int16)


# Cleanup - drop redundant columns

df=df.drop(['d','date','weekday'],axis=1)


return df

```

In [16]:

```

# Process data by store, write to disk for further steps

sales_train_evaluation_ = reduce_mem_usage(sales_train_evaluation_)

for store_id in STORES_IDS:

    x = sales_train_evaluation_[sales_train_evaluation_.store_id == store_id]

    df = data_processing(x)

    file_path = '/content/'+store_id+'.csv'

    df.to_csv(file_path, index=False)

```

Mem. usage decreased to 96.13 Mb (0.0% reduction)

In [34]:

```

# Define loss function

def tweedie_eval(y_pred, y_true, p=1.5):

    y_true = y_true.get_label()

    a = y_true*np.exp(y_pred, (1-p)) / (1-p)

```

```

        b = np.exp(y_pred, (2-p))/(2-p)

        loss = -a + b

    return loss

```

```

def custom_split(X, y, groups):
    for train_index, test_index in groups:
        original_train_index = np.array(X[X['day'].isin(train_index+1)].index)
        original_test_index = np.array(X[X['day'].isin(test_index+1)].index)
        yield original_train_index, original_test_index

tweedie = make_scorer(tweedie_eval)

```

In [50]:

```
# Hyperparameter tuning with RandomizedSearchCV
```

```
from scipy.stats import reciprocal
```

```
for store_id in STORES_IDS:
```

```
    file_path = '/content/'+store_id+'.csv'
```

```
    df1 = pd.read_csv(file_path)
```

```
    df1 = reduce_mem_usage(df1)
```

```
    # remove testing data
```

```
    df1 = df1[df1['day']<1942]
```

```
    df = df1.reset_index().drop("index",axis=1)
```

```
    X = df.iloc[:, 1:]
```

```
    y = df['demand']
```

```
tscv = TimeSeriesSplit(n_splits=5, test_size=28)
```

```
groups = tscv.split(df1['day'].unique())
```

```
lgb = LGBMRegressor( objective='tweedie')
```

```
lgb_grid = {
    'tweedie_variance_power': [1.1],

```

```
lgb_reg = RandomizedSearchCV(lgb, param_distributions=lgb_grid,
                             n_jobs=-1, scoring='neg_root_mean_squared_error', cv=custom_split(X, y, groups))
lgb_reg.fit(X, y)

print(store_id, 't', lgb_reg.best_params_, lgb_reg.best_score_)

model_path = '/content/lgb_model_'+store_id+'.bin'

pickle.dump(lgb_reg.best_estimator_, open(model_path, 'wb'))
```

Out[50]:

CA 1 ma-2.30774943648447

Out[50]:

```
RandomizedSearchCV(cv=<generator object custom_split at 0x7fe475194750>,
  estimator=LGBMRegressor(objective='tweddie'), n_jobs=-1,
  param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe476381490>,
    'max_depth': [50, 51, 52, 53, 54, 55,
      56, 57, 58, 59, 60, 61,
      62, 63, 64, 65, 66, 67,
      68, 69],
    'n_estimators': [100, 110, 120, 130,
      140, 150, 160, 170,
      180, 190, 200, 210,
      220, 230, 240, 250,
```

```

                260, 270, 280, 290],
        'num_leaves': [150, 151, 152, 153, 154,
                        155, 156, 157, 158, 159,
                        160, 161, 162, 163, 164,
                        165, 166, 167, 168, 169,
                        170, 171, 172, 173, 174,
                        175, 176, 177, 178, 179, ...],
        'tweedie_variance_power': [1.1]},
    scoring='neg_root_mean_squared_error')
CA_2    {'learning_rate': 0.051583904830467577, 'max_depth': 53, 'n_estimators': 280, 'num_leaves': 222,
'tweedie_variance_power': 1.1} -2.144873011113553
Mem. usage decreased to 206.11 Mb (82.7% reduction)
Out[50]:
RandomizedSearchCV(cv=<generator object custom_split at 0x7fe476573050>,
    estimator=LGBMRegressor(objective='tweedie'), n_jobs=-1,
    param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe476913d50>,
        'max_depth': [50, 51, 52, 53, 54, 55,
                        56, 57, 58, 59, 60, 61,
                        62, 63, 64, 65, 66, 67,
                        68, 69],
        'n_estimators': [100, 110, 120, 130,
                          140, 150, 160, 170,
                          180, 190, 200, 210,
                          220, 230, 240, 250,
                          260, 270, 280, 290],
        'num_leaves': [150, 151, 152, 153, 154,
                        155, 156, 157, 158, 159,
                        160, 161, 162, 163, 164,
                        165, 166, 167, 168, 169,
                        170, 171, 172, 173, 174,
                        175, 176, 177, 178, 179, ...],
        'tweedie_variance_power': [1.1]},
    scoring='neg_root_mean_squared_error')
CA_3    {'learning_rate': 0.07582967729206709, 'max_depth': 54, 'n_estimators': 270, 'num_leaves': 208,
'tweedie_variance_power': 1.1} -3.124439480568764
Mem. usage decreased to 206.11 Mb (82.7% reduction)
Out[50]:
RandomizedSearchCV(cv=<generator object custom_split at 0x7fe475194950>,
    estimator=LGBMRegressor(objective='tweedie'), n_jobs=-1,
    param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe47593a8d0>,
        'max_depth': [50, 51, 52, 53, 54, 55,
                        56, 57, 58, 59, 60, 61,
                        62, 63, 64, 65, 66, 67,
                        68, 69],
        'n_estimators': [100, 110, 120, 130,
                          140, 150, 160, 170,
                          180, 190, 200, 210,
                          220, 230, 240, 250,
                          260, 270, 280, 290],
        'num_leaves': [150, 151, 152, 153, 154,
                        155, 156, 157, 158, 159,
                        160, 161, 162, 163, 164,
                        165, 166, 167, 168, 169,
                        170, 171, 172, 173, 174,

```



```
175, 176, 177, 178, 179, ...],
'tweedie_variance_power': [1.1]},
scoring='neg_root_mean_squared_error')
CA_4    {'learning_rate': 0.1478995769975061, 'max_depth': 62, 'n_estimators': 130, 'num_leaves': 299, 'tweedie_variance_power':
1.1} -1.4947936396158847
```

Mem. usage decreased to 206.11 Mb (82.7% reduction)

Out[50]:

```
RandomizedSearchCV(cv=<generator object custom_split at 0x7fe476492850>,
estimator=LGBMRegressor(objective='tweedie'), n_jobs=-1,
param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe476821390>,
'max_depth': [50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67,
68, 69],
'n_estimators': [100, 110, 120, 130,
140, 150, 160, 170,
180, 190, 200, 210,
220, 230, 240, 250,
260, 270, 280, 290],
'num_leaves': [150, 151, 152, 153, 154,
155, 156, 157, 158, 159,
160, 161, 162, 163, 164,
165, 166, 167, 168, 169,
170, 171, 172, 173, 174,
175, 176, 177, 178, 179, ...],
'tweedie_variance_power': [1.1]},
scoring='neg_root_mean_squared_error')
```

```
TX_1    {'learning_rate': 0.16804816033695238, 'max_depth': 53, 'n_estimators': 150, 'num_leaves': 192,
'tweedie_variance_power': 1.1} -1.9312170398919613
```

Mem. usage decreased to 206.11 Mb (82.7% reduction)

Out[50]:

```
RandomizedSearchCV(cv=<generator object custom_split at 0x7fe475194350>,
estimator=LGBMRegressor(objective='tweedie'), n_jobs=-1,
param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe47655e490>,
'max_depth': [50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67,
68, 69],
'n_estimators': [100, 110, 120, 130,
140, 150, 160, 170,
180, 190, 200, 210,
220, 230, 240, 250,
260, 270, 280, 290],
'num_leaves': [150, 151, 152, 153, 154,
155, 156, 157, 158, 159,
160, 161, 162, 163, 164,
165, 166, 167, 168, 169,
170, 171, 172, 173, 174,
175, 176, 177, 178, 179, ...],
'tweedie_variance_power': [1.1]},
scoring='neg_root_mean_squared_error')
```

```
TX_2    {'learning_rate': 0.036646621908299024, 'max_depth': 50, 'n_estimators': 270, 'num_leaves': 257,
'tweedie_variance_power': 1.1} -2.184581615599202
```

Mem. usage decreased to 206.11 Mb (82.7% reduction)

Out[50]:

```
RandomizedSearchCV(cv=<generator object custom_split at 0x7fe4634cded0>,
  estimator=LGBMRegressor(objective='tweedie'), n_jobs=-1,
  param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe4752718d0>,
    'max_depth': [50, 51, 52, 53, 54, 55,
      56, 57, 58, 59, 60, 61,
      62, 63, 64, 65, 66, 67,
      68, 69],
    'n_estimators': [100, 110, 120, 130,
      140, 150, 160, 170,
      180, 190, 200, 210,
      220, 230, 240, 250,
      260, 270, 280, 290],
    'num_leaves': [150, 151, 152, 153, 154,
      155, 156, 157, 158, 159,
      160, 161, 162, 163, 164,
      165, 166, 167, 168, 169,
      170, 171, 172, 173, 174,
      175, 176, 177, 178, 179, ...],
    'tweedie_variance_power': [1.1]},
  scoring='neg_root_mean_squared_error')
TX_3    {'learning_rate': 0.16132262516317375, 'max_depth': 59, 'n_estimators': 240, 'num_leaves': 154,
'tweedie_variance_power': 1.1} -2.1470173603077294
Mem. usage decreased to 206.11 Mb (82.7% reduction)
```

Out[50]:

```
RandomizedSearchCV(cv=<generator object custom_split at 0x7fe475194e50>,
  estimator=LGBMRegressor(objective='tweedie'), n_jobs=-1,
  param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe476459ad0>,
    'max_depth': [50, 51, 52, 53, 54, 55,
      56, 57, 58, 59, 60, 61,
      62, 63, 64, 65, 66, 67,
      68, 69],
    'n_estimators': [100, 110, 120, 130,
      140, 150, 160, 170,
      180, 190, 200, 210,
      220, 230, 240, 250,
      260, 270, 280, 290],
    'num_leaves': [150, 151, 152, 153, 154,
      155, 156, 157, 158, 159,
      160, 161, 162, 163, 164,
      165, 166, 167, 168, 169,
      170, 171, 172, 173, 174,
      175, 176, 177, 178, 179, ...],
    'tweedie_variance_power': [1.1]},
  scoring='neg_root_mean_squared_error')
WI_1    {'learning_rate': 0.04500479865741088, 'max_depth': 64, 'n_estimators': 220, 'num_leaves': 225,
'tweedie_variance_power': 1.1} -1.8347939897611092
Mem. usage decreased to 206.11 Mb (82.7% reduction)
```

Out[50]:

```
RandomizedSearchCV(cv=<generator object custom_split at 0x7fe475194650>,
  estimator=LGBMRegressor(objective='tweedie'), n_jobs=-1,
  param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe474e74310>,
    'max_depth': [50, 51, 52, 53, 54, 55,
```

```

56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67,
68, 69],
'n_estimators': [100, 110, 120, 130,
140, 150, 160, 170,
180, 190, 200, 210,
220, 230, 240, 250,
260, 270, 280, 290],
'num_leaves': [150, 151, 152, 153, 154,
155, 156, 157, 158, 159,
160, 161, 162, 163, 164,
165, 166, 167, 168, 169,
170, 171, 172, 173, 174,
175, 176, 177, 178, 179, ...],
'tweedie_variance_power': [1.1]},
scoring='neg_root_mean_squared_error')
WI_2    {'learning_rate': 0.0612933414734137, 'max_depth': 50, 'n_estimators': 290, 'num_leaves': 200, 'tweedie_variance_power':
1.1} -3.285028287799137
Mem. usage decreased to 206.11 Mb (82.7% reduction)

```

Out[50]:

```

RandomizedSearchCV(cv=<generator object custom_split at 0x7fe475194050>,
estimator=LGBMRegressor(objective='tweedie'), n_jobs=-1,
param_distributions={'learning_rate': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fe475608110>,
'n_estimators': [100, 110, 120, 130,
140, 150, 160, 170,
180, 190, 200, 210,
220, 230, 240, 250,
260, 270, 280, 290],
'num_leaves': [150, 151, 152, 153, 154,
155, 156, 157, 158, 159,
160, 161, 162, 163, 164,
165, 166, 167, 168, 169,
170, 171, 172, 173, 174,
175, 176, 177, 178, 179, ...],
'tweedie_variance_power': [1.1]},
scoring='neg_root_mean_squared_error')
WI_3    {'learning_rate': 0.05896493503811511, 'max_depth': 65, 'n_estimators': 160, 'num_leaves': 203,
'tweedie_variance_power': 1.1} -2.352375351361978

```

In [51]:

```

# Initialize submission dataframe

pred_test=pd.DataFrame()

pred_test['id']=sales_train_evaluation['id']

pred_test['store_id']=sales_train_evaluation['store_id']

for i in range(1,29):

    pred_test['F'+str(i)]=np.nan

    pred_test['F'+str(i)]=pred_test['F'+str(i)].astype(np.float16)

```

In [52]:

```
# Make test predictions by store

for store_id in STORES_IDS:

    file_path = '/content/'+store_id+'.csv'

    df = pd.read_csv(file_path)

    x_test=df.loc[df['day']>=1942]

    x_test = x_test.drop(['demand'],axis=1)

    model_path = '/content/lgb_model_'+store_id+'.bin'

    lgb = pickle.load(open(model_path, 'rb'))

    k=1

    for i in range(1942,1970):

        # Read all our models and make predictions for each day/store pairs

        pred_test['F'+str(k)][pred_test['store_id']==store_id]=lgb.predict(x_test[x_test['day']==(i)])

        k+=1

prediction_test = np.round(pred_test,2)
```

In [53]:

```
# Output the final submission file

import time

current_timestamp = int(time.time())

prediction_test = prediction_test.drop('store_id',axis=1)

sample_submission = pd.read_csv('/content/sample_submission.csv')

sample_validation = sample_submission.iloc[:30490,: ]

final = pd.concat([sample_validation, prediction_test])

file_path = '/content/prediction_result' + str(current_timestamp) + '.csv'

final.to_csv(file_path,index=False)
```

In [54]:

```
# ##### TEST Predict #####

# df=df.drop(['demand'],axis=1)

# #Loading Already Trained LightGBM Regressor Model for Computaion

# with open('/content/lgb_model.pkl','rb') as f:

#     lgb=pickle.load(f)
```

```

# pred_test=pd.DataFrame()
# pred_test['id']=x['id']
# j=1
# k=1
# for i in range(1942,1970):
#     pred_test['F'+str(k)]=lgb.predict(x_test[x_test['day']==(i)])
#     k+=1

# prediction_test = np.round(pred_test,2)

## Loop over each prediction day
## As rolling lags are the most timeconsuming
## we will calculate it for whole day
# for PREDICT_DAY in range(1,29):
#     print('Predict | Day:', PREDICT_DAY)

#     for store_id in STORES_IDS:

#         # Read all our models and make predictions
#         # for each day/store pairs
#         model_path = '/content/lgb_model_'+store_id+'.bin'

#         estimator = pickle.load(open(model_path, 'rb'))

#         day_mask = base_test['d']==(END_TRAIN+PREDICT_DAY)
#         store_mask = base_test['store_id']==store_id

#         mask = (day_mask)&(store_mask)
#         base_test[TARGET][mask] = estimator.predict(grid_df[mask][MODEL_FEATURES])

#         # Make good column naming and add
#         # to all_preds DataFrame
#         temp_df = base_test[day_mask][['id',TARGET]]
#         temp_df.columns = ['id','F'+str(PREDICT_DAY)]

```

```
#         if 'id' in list(all_preds):
#             all_preds = all_preds.merge(temp_df, on=['id'], how='left')
#         else:
#             all_preds = temp_df.copy()

#         print('#*10, ' %0.2f min round |' % ((time.time() - start_time) / 60),
#               ' %0.2f min total |' % ((time.time() - main_time) / 60),
#               ' %0.2f day sales |' % (temp_df['F'+str(PREDICT_DAY)].sum()))
#         del temp_df

# all_preds = all_preds.reset_index(drop=True)
# all_preds
```

X. Reference & Credit

<https://www.kaggle.com/c/m5-forecasting-accuracy/discussion/163216>

<https://www.kaggle.com/c/m5-forecasting-accuracy/discussion/164599>

<https://dipanshurana.medium.com/m5-forecasting-accuracy-1b5a10218fcf>

<https://www.kaggle.com/competitions/m5-forecasting-accuracy/discussion/150614>