Time Series with LGBM

October 10, 2020

0.1 Time Series Analysis with LightGBM

Question: how could you estimate the point forecasts of the unit sales of products at Walmart in the U.S.?

Input datasets are from Kaggle. I estimated item sales at stores in various locations for two 28-day time periods.

```
In [ ]: #import libraries
        import os
        import gc
        import warnings

        import pandas as pd
        from pandas.plotting import register_matplotlib_converters
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        import lightgbm as lgb
        from sklearn.metrics import mean_squared_error
        from sklearn.preprocessing import LabelEncoder
        from sklearn.preprocessing import OneHotEncoder

        warnings.filterwarnings("ignore")
        pd.set_option("display.max_columns", 500)
        pd.set_option("display.max_rows", 500)
        register_matplotlib_converters()
        sns.set()
```

0.1.1 Updates

—Used One-Hot Encoder in feature engineering, as Label Encoder assigned an abstract "order" to categorical features

0.1.2 Helper functions and Input Dataframes

```
In [1]: #define a helper function to reduce memory sizes as the original datasets are large
        def reduce_mem_usage(df, verbose=False):
            start_mem = df.memory_usage().sum() / 1024 ** 2
```

```
    int_columns = df.select_dtypes(include=["int"]).columns
    float_columns = df.select_dtypes(include=["float"]).columns

    for col in int_columns:
        df[col] = pd.to_numeric(df[col], downcast="integer")

    for col in float_columns:
        df[col] = pd.to_numeric(df[col], downcast="float")

    end_mem = df.memory_usage().sum() / 1024 ** 2
    if verbose:
        print(
            "Mem. usage decreased to {:5.2f} Mb ({:.1f}% reduction)".format(
                end_mem, 100 * (start_mem - end_mem) / start_mem
            )
        )
    return df
```

```
In [ ]: #define a function to read in data and reduce memory size
        def read_data():
            calendar = pd.read_csv("/kaggle/input/m5-forecasting-accuracy/calendar.csv").pipe(r
            prices = pd.read_csv("/kaggle/input/m5-forecasting-accuracy/sell_prices.csv").pipe
            sales = pd.read_csv("/kaggle/input/m5-forecasting-accuracy/sales_train_evaluation.c
            sample_eval = pd.read_csv("/kaggle/input/m5-forecasting-accuracy/sample_submission

            return sales, prices, calendar, sample_eval
```

```
In [ ]: #read in three original datasets and reduce sizes with pipe
        sales, prices, calendar, sample_eval= read_data()
        #check how many items to predict
        NUM_ITEMS = sales.shape[0]  # 30490
        #predicting 28 days of sales
        DAYS_PRED = 28
```

```
In [ ]: #convert categorical vars to numerical vars
        enc = preprocessing.OneHotEncoder()
        le = preprocessing.LabelEncoder()
        def encode_categorical(df, cols):
            for col in cols:
                not_null = df[col][df[col].notnull()]
                df[col] = le.fit_transform(not_null)
                df[col] = pd.Series(enc.fit_transform(df[col]), index=not_null.index)

            return df

        #apply one-hot encoding to event events/types in calendar data
        calendar = encode_categorical(
                                    calendar, ["event_name_1", "event_type_1", "event_name_2",
```

```
                                    ).pipe(reduce_mem_usage)

        #apply one-hot encoding to ids in sales data
        sales = encode_categorical(
                                sales, ["item_id", "dept_id", "cat_id", "store_id", "state_id"]
                                 ).pipe(reduce_mem_usage)

        prices = encode_categorical(prices, ["item_id", "store_id"]).pipe(reduce_mem_usage)
```

In [ ]:
```
#define a helper function to extract digits from date variables
def extract_num(ser):
    return ser.str.extract(r"(\d+)").astype(np.int16)

#concatenate train and evaluation data
#extract ids and generate predicting ids
def reshape_sales(sales, submission, d_thresh=0, verbose=True):
    id_columns = ["id", "item_id", "dept_id", "cat_id", "store_id", "state_id"]
    #add evaluation days to the end of ids
    #d1 is the number of units sold on day 1
    evals_columns = ["id"] + [f"d_{d}" for d in range(1942, 1942 + DAYS_PRED)]
    #separate test data: return if id variable ends with evaluation
    evals = sample_eval[sample_eval["id"].str.endswith("evaluation")]

    #generate a table with ids. unpivot sales data, and a col of eval ids
    product = sales[id_columns]
    sales = sales.melt(id_vars=id_columns, var_name="d", value_name="demand")
    sales = reduce_mem_usage(sales)
    evals.columns = evals_columns

    #merge evals with product table--a table of ids
    evals = evals.merge(product, how="left", on="id")
    #unpivot
    evals = evals.melt(id_vars=id_columns, var_name="d", value_name="demand")

    #claim train and evaluation data then stack together
    sales["part"] = "train"
    evals["part"] = "evaluation"
    data = pd.concat([sales, evals], axis=0)

    #extract id numbers and drop data out of range
    data["d"] = extract_num(data["d"])
    data = data[data["d"] >= d_thresh]

    return data
```

In [ ]:
```
#drop redundant time variables and add the rest to the main dataframe--only keep days 
def merge_calendar(data, calendar):
    calendar = calendar.drop(["weekday", "wday", "month", "year"], axis=1)
```

```
        return data.merge(calendar, how="left", on="d")

    #add price variables to the main dataframe
    def merge_prices(data, prices):
        return data.merge(prices, how="left", on=["store_id", "item_id", "wm_yr_wk"])
```

```
In [ ]: #create a concatenated dataframe with the most recent 2 years
        data = reshape_sales(sales, sample_eval, d_thresh=1941 - int(365 * 2))
        #only keep digit parts of time variables
        calendar["d"] = extract_num(calendar["d"])
        #add calendar variables to the main data frame
        data = merge_calendar(data, calendar)
        #add prices to the main dataframe
        data = merge_prices(data, prices)
        #reduce memory usage
        data = reduce_mem_usage(data)
```

0.1.3 Feature Engineering

```
In [ ]: #create demand features: rolling window summary statistics
        def add_demand_features(df):
            #28 days to predict
            for diff in [0, 28]:
                shift = DAYS_PRED + diff
                #shift index by 28+diff
                df[f"shift_t{shift}"] = df.groupby(["id"])["demand"].transform(
                    lambda x: x.shift(shift)
                )

            diff = 28
            #roll features: summary statistics
            for window in [28, 56, 84]:
                df[f"shift_t{diff}_rolling_std_t{window}"] = df.groupby(["id"])["demand"].trans
                    lambda x: x.shift(diff).rolling(window).std()
                )
                df[f"shift_t{diff}_rolling_mean_t{window}"] = df.groupby(["id"])["demand"].tra
                    lambda x: x.shift(diff).rolling(window).mean()
                )
                df[f"rolling_min_t{window}"] = df.groupby(["id"])["demand"].transform(
                    lambda x: x.shift(diff).rolling(window).min()
                )
                df[f"rolling_max_t{window}"] = df.groupby(["id"])["demand"].transform(
                    lambda x: x.shift(diff).rolling(window).max()
                )
                df[f"rolling_sum_t{window}"] = df.groupby(["id"])["demand"].transform(
                    lambda x: x.shift(diff).rolling(window).sum()
                )
            #measure asymmetry
```

```
    df["rolling_skew_t56"] = df.groupby(["id"])["demand"].transform(
        lambda x: x.shift(DAYS_PRED).rolling(56).skew()
    )
    #measure tailedness
    df["rolling_kurt_t56"] = df.groupby(["id"])["demand"].transform(
        lambda x: x.shift(DAYS_PRED).rolling(56).kurt()
    )
    return df
```

In [ ]:
```
#create price features by setting up rolling window summary statistics
def add_price_features(df):
    df["shift_price_t1"] = df.groupby(["id"])["sell_price"].transform(
        lambda x: x.shift(1)
    )
    df["price_change_t1"] = (df["shift_price_t1"] - df["sell_price"]) / (
        df["shift_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_t30"] = df.groupby(["id"])["sell_price"].transform(
        lambda x: x.rolling(30).std()
    )
    return df.drop(["rolling_price_max_t365", "shift_price_t1"], axis=1)
```

In [ ]:
```
#extract time features
def add_time_features(df, dt_col):
    #convert to datetime varible on the purpose of attribute extraction
    df[dt_col] = pd.to_datetime(df[dt_col])
    attrs = [
            "year",
            "quarter",
            "month",
            "week",
            "day",
            "dayofweek"
            ]

    for attr in attrs:
        #save space
        dtype = np.int16 if attr == "year" else np.int8
        #get the value of attribute of time varible
        #dt can be used to access the values of the series as datetimelike and return
        df[attr] = getattr(df[dt_col].dt, attr).astype(dtype)
    #additional time variables that require manipulation
```

```
        df["is_weekend"] = df["dayofweek"].isin([5, 6]).astype(np.int8)
        return df
```

In [ ]:
```
#apply helper func to create features
data = add_demand_features(data).pipe(reduce_mem_usage)
data = add_price_features(data).pipe(reduce_mem_usage)
data = add_original_features(data).pipe(reduce_mem_usage)
dt_col = "date"
data = add_time_features(data, dt_col).pipe(reduce_mem_usage)
data = data.sort_values("date")
```

0.1.4 Training Model

In [ ]:
```
#sklearn.model_selection.TimeSeriesSplit doesn't work,
#as it only has two parameters: n_split and max size of a single training set
#I want to specify the days being tested: 28
#create a customized time series splitter instead

#seconds in a day--used in modeling iteratively
SEC_IN_DAY = 3600 * 24
class CustomTimeSeriesSplitter:
    #initialize macro variables; col name of date is "d"
    def __init__(self, n_splits, train_days, test_days, day_col):
        self.n_splits = n_splits
        #train_days in each split
        self.train_days = train_days
        #test days in each split
        self.test_days = test_days
        self.day_col = day_col

    #specify the beginning and end index values of each fold
    #X will be X_train, y will be y_train
    def split(self, X, y):
        #calculate the time range in seconds wrt the first row (initial time point)
        sec = (X[self.day_col] - X[self.day_col].iloc[0]) * SEC_IN_DAY
        duration = sec.max()
        #total secs of days in trainning and test sets
        train_sec = self.train_days * SEC_IN_DAY
        test_sec = self.test_days * SEC_IN_DAY
        total_sec = test_sec + train_sec

        #cross validation setup
        #window size = prediction size
        step = DAYS_PRED * SEC_IN_DAY
        #for each split: 0, 1, 2, 3, 4
        for idx in range(self.n_splits):
            #compute rolling train data for each split; test data follows 28 days late
            shift = (self.n_splits - (idx + 1)) * step
```

```
                #minus train and test data from the last split
                train_start = duration - total_sec - shift
                #365 days forwards
                train_end = train_start + train_sec
                #28 days
                test_end = train_end + test_sec
                #For each split: identify if training data has a reasonable starting and e
                train_mask = (sec > train_start) & (sec <= train_end)
                #For each split: identify if test data has a reasonable starting and end p
                #if not the last split, identify test data that was after trainning data a
                if idx != self.n_splits - 1:
                    test_mask = (sec > train_end) & (sec <= test_end)
                #if the last split, don't need to check the end point
                else:
                    test_mask = sec > train_end
                #use yield as I'm generating index values in each iteration
                #will be used to print the min d var of train and test (begining and end o
                yield sec[train_mask].index.values, sec[test_mask].index.values
```

In [ ]:
```
#build a dictionary of parameters
cv_params = {
    "n_splits": 5,
    "train_days": 365,
    "test_days": 28,#DAYS_PRED,
    "day_col": "d",
}
#use ** to call dictionary values
cv = CustomTimeSeriesSplitter(**cv_params)
```

In [ ]:
```
#selected part of features by experimentation
features = [
    "item_id",
    "dept_id",
    "cat_id",
    "store_id",
    "state_id",
    "event_name_1",
    "event_type_1",
    "event_name_2",
    "event_type_2",
    "snap_CA",
    "snap_TX",
    "snap_WI",
    "sell_price",
    # demand features
    "shift_t28",
    "shift_t56",
    # std
```

```
            "shift_t28_rolling_std_t28",
            "shift_t28_rolling_std_t56",
            "shift_t28_rolling_std_t84",
            # mean
            "shift_t28_rolling_mean_t28",
            "shift_t28_rolling_mean_t56",
            "shift_t28_rolling_mean_t84",
            # min,
            "rolling_min_t28",
            # max
            "rolling_max_t28",
            "rolling_max_t56",
            # sum
            "rolling_sum_t28",
            "rolling_sum_t56",
            "rolling_kurt_t28",
            "price_change_t365",
            "rolling_price_std_t30",
            # time features
            "year",
            "quarter",
            "month",
            "week",
            "day",
            "dayofweek",
            "is_weekend"
            ]
```

In [ ]:
```
#separate training data and testing data
#training data: days before d1942
is_train = (data["d"] < 1942)
#test data: days after d1942
is_test = (data["d"] >= 1942)

# Creating training data: "d" (id variable from the original dataset) and selected fea
X_train = data[is_train][[day_col] + features].reset_index(drop=True)
y_train = data[is_train]["demand"].reset_index(drop=True)
# Creating test data: selected features
X_test = data[is_test][features].reset_index(drop=True)
```

In [ ]:
```
#Tune Hyperparameters
#use poisson objective function
bst_params =  {'lambda_l1': 0.0002,
               'lambda_l2': 9.2425e-07,
               'num_leaves': 31,
               'feature_fraction': 0.584,
               'bagging_fraction': 1.0,
               'bagging_freq': 0,
```

```
            'min_child_samples': 20,
            'boosting_type': 'gbdt',
            'metric': 'rmse',
            'objective': 'poisson',
            'n_jobs': -1,
            'seed': 42,
            'learning_rate': 0.03,
            'min_data_in_leaf': 20}

#define models
def train_lgb(bst_params, fit_params, X, y, cv, drop_when_train):
    models = []

    for idx_fold, (idx_trn, idx_val) in enumerate(cv.split(X, y)):
        #print fold index / total number of folds
        print(f"\n----- Fold: ({idx_fold + 1} / {cv.get_n_splits()}) -----\n")

        #extract reasonable training and validating data (within the expected range)
        X_trn, X_val = X.iloc[idx_trn], X.iloc[idx_val]
        y_trn, y_val = y.iloc[idx_trn], y.iloc[idx_val]
        #print the min d var of train and test (begining and end of validating data)
        print(f'\n train d min: {X_trn["d"].min()} \n valid d min: {X_val["d"].min()} \

        #initialize train dataset
        train_set = lgb.Dataset(
                                X_trn.drop(drop_when_train, axis=1), #main dataset to
                                label=y_trn, #label of dataset
                                categorical_feature=["item_id"], #list of int: indices
                                )
        #initialize validation dataset
        val_set = lgb.Dataset(
                                X_val.drop(drop_when_train, axis=1),
                                label=y_val,
                                categorical_feature=["item_id"],
                             )

        #perform training with parameters
        model = lgb.train(
                            bst_params, #given parameters
                            train_set, #data to be trained on
                            valid_sets=[train_set, val_set], #data to be evaluated dur
                            valid_names=["train", "valid"], #names of valid sets
                            evals_result=eval_result, #a dictionary to store all the e
                            num_boost_round= 100_000, #number of boosting iterations: 
                            early_stopping_rounds= 100, #validation score needs to imp
                            verbose_eval= 100, #int: the evaluation metric is printed 
                            )
        #stack validated models
```

```
        models.append(model)
        #clean subset of data for the next fold
        del idx_trn, idx_val, X_trn, X_val, y_trn, y_val
        gc.collect()

    return models, eval_result
```

```
In [ ]: #output models and evaluation
        models, evals = train_lgb(
                                bst_params, fit_params, X_train, y_train, cv, drop_when_train=
                                )
```

0.1.5 prediction

I'm using a weighted average meta model: put the highest weight on the most recent timeseries model(fold 5) and put smaller weights on other folds according to evaluation results

```
In [ ]: preds = np.zeros(X_test.shape[0])
        preds = 0.6*models[4].predict(X_test)+0.3*models[1].predict(X_test)+0.1*models[3].predi
```