Predicting Units of Product Sold (Modeling)

In [1]:
```
# imports for notebook
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import OneHotEncoder, MultiLabelBinarizer
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from lightgbm import LGBMRegressor
from xgboost import XGBRegressor
```

Dates Data

In [2]:
```
# read in the dates and their recorded unique characteristics
dates = pd.read_csv('C:/Users/TWood/Downloads/m5-forecasting-accuracy/calenda
```

In [3]:
```
# take a look at the dates df
dates.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1969 entries, 0 to 1968
Data columns (total 14 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   date          1969 non-null   datetime64[ns]
 1   wm_yr_wk      1969 non-null   int64 
 2   weekday       1969 non-null   object
 3   wday          1969 non-null   int64 
 4   month         1969 non-null   int64 
 5   year          1969 non-null   int64 
 6   d             1969 non-null   object
 7   event_name_1  162 non-null    object
 8   event_type_1  162 non-null    object
 9   event_name_2  5 non-null      object
 10  event_type_2  5 non-null      object
 11  snap_CA       1969 non-null   int64 
 12  snap_TX       1969 non-null   int64 
 13  snap_WI       1969 non-null   int64 
```

In [4]:
```
# most days have no event, replace NaN with "None"
dates.replace(np.NaN, 'None', inplace=True)
```

Binarizing Events Columns

Some days have multiple events. One hot encoding will not be able to represent columns with

multiple events, so the information will need to be in a format that the MultiLabelBinarizer can use. I'll make a new column that contains a list of all events on a given day.

In [5]:
```
# remove spaces from all the events names
dates['event_name_1'] = dates['event_name_1'].str.replace(' ', '')
dates['event_name_2'] = dates['event_name_2'].str.replace(' ', '')
```

In [6]:
```
# create event column that contains a string of both events with a space betw
dates['event'] = dates['event_name_1'] + ' ' + dates['event_name_2']
```

In [7]:
```
# split will turn the string into a list of both events
dates['event'] = dates['event'].str.split()
```

In [8]:
```
# removes the second element from the list when it is None
dates['event'] = dates['event'].apply(lambda x: [x[0]] if x[1] == 'None' else
```

In [9]:
```
# instantiat the MultiLabelBinarizer and fit it to the event column
mlb = MultiLabelBinarizer()
mlb.fit(dates['event'])
values = pd.DataFrame(mlb.transform(dates['event']), columns=mlb.classes_)
```

In [10]:
```
# adds the encoded columns to the dates dataframe
dates = pd.concat([dates, values], axis=1)
```

In [11]:
```
# drops the redundant event columns from dates
dates.drop(columns=['event_name_1', 'event_type_1', 'event_name_2', 'event_ty
```

In [12]:
```
# instantiate the OneHotEncoder and fit it to weekday and month columns
ohe = OneHotEncoder(sparse=False)
ohe.fit(dates[['weekday', 'month']])
values = ohe.transform(dates[['weekday', 'month']])
values = pd.DataFrame(values, columns=ohe.get_feature_names())
```

In [13]:
```
# add the one hot encoded columns to the dates dataframe
dates = pd.concat([dates, values], axis=1)
```

In [14]:
```
# drop the unnecessary columns from the dates dataframe
dates.drop(columns=['wday', 'year', 'month', 'weekday', 'snap_WI', 'snap_TX']
```

Units Sold Data

This csv contains much more information than the calendar csv, and it is still in a wide format. It will need to be melted to be used in the model, and once this dataframe is merged with the other two dataframes, it will take up a huge amount of RAM. In order to work with the data on a computer

with only 8GB of RAM, I've limited the data to only the Foods 1 department of the first California store. I'll provide notebooks for other stores and departments in the modeling folder on the repository.

In [15]:
```
# read in the data for units sold
val = pd.read_csv('C:/Users/TWood/Downloads/m5-forecasting-accuracy/sales_tra
```

In [16]:
```
# select only the data from department FOODS_1 in store CA_1
CA1_F1 = val[(val['store_id'] == 'CA_1')&(val['dept_id'] == 'FOODS_1')]
```

In [17]:
```
# reducing the unnecessary columns to make the melt faster
CA1_F1.drop(columns=['item_id', 'dept_id', 'cat_id', 'store_id', 'state_id'],
```

```
C:\Users\TWood\anaconda3\envs\learn-env\lib\site-packages\pandas\core\fram
e.py:4163: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-doc
s/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://p
andas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-vi
ew-versus-a-copy)
  return super().drop(
```

In [18]:
```
# convert the dataframe into long format
CA1_F1_ts = CA1_F1.melt(id_vars=['id'], var_name='d', value_name='sales')
```

Prices Data

In [19]:
```
# read in the data on the prices for the products
prices = pd.read_csv('C:/Users/TWood/Downloads/m5-forecasting-accuracy/sell_p
```

In [20]:
```
# make a new id column that matches the format of the id column in CA1_F1_ts
prices['id'] = prices['item_id'] + '_' + prices['store_id'] + '_validation'
```

Modeling DataFrame

The final data frame is only going to contain information from the FOODS 1 department of the first California store. Creating the data frame from merging all three csv's makes an extremely large dataframe that wouldn't fit on computers with only 8GB of RAM. I decided it would be more feasible to limit my model to only the first California score. Even only looking at the one store quickly becomes unwieldy, so we'll find the best algorithm and tune it with only the FOODS 1 department. After the best model is found, another couple notebooks will fit the model again and give the forecasts for the rest of items in the store. A little cleaning will be needed for the resulting merged CA1_F1_price, and a few time series features will be added to the dataframe, then the CA1_F1_price will be good for modeling.

In [21]:
```
# create a dataframe with the information from all dataframes
CA1_F1_price = CA1_F1_ts.merge(dates, on='d').merge(prices.drop(columns=['stc
```

In [22]:
```
# set the index of the dataframe to the date column
CA1_F1_price.set_index('date', inplace=True)
```

Missing Prices

There are several missing prices in the dataframe, which could cause a big problem when modeling. However, upon further inspection, there were always zero sales of a product when it the products price was missing. This suggests that the price was not entered for these weeks because it was not on the shelves for those weeks. The weeks where the price is missing can easily be dropped when fitting the model as they will not be representative of the actual demand of the product. Furthermore, all 30490 distinct ids are present during the last week 11621, which means that every product in every store was on the shelf. If any of these prices were missing, it would be best to remove these predictions from the model.

In [23]:
```
# there are several nulls after left joining with prices
CA1_F1_price['sell_price'].isna().sum()
```

Out[23]: 61292

In [24]:
```
# there are never any sales of a product when its price is missing for the we
((CA1_F1_price['sales'] == 0)&(CA1_F1_price['sell_price'].isna())).sum()
```

Out[24]: 61292

In [25]:
```
# 30490 total items
len(val)
```

Out[25]: 30490

In [26]:
```
# the sell prices are present for all 30490 items for the final week, matches
(prices['wm_yr_wk'] == 11621).sum()
```

Out[26]: 30490

Lag Columns

This lag column will tell the model how many units were sold 28 days earlier. In order for this model to be maximally useful, Wal-Mart will need to be able to predict sales of an item much further than a week before hand. My original model contained several more lags from within 28 days, but was unable to predict days further out. It could be used to predict the next day accurately, but had a serious data leakage issue when forecasting more than one day out, as it used lags that a real model would not have access to. One could create a model that forecasts one day at a time

replacing the lags with forecasted lags, but the model would suffer from giving too much weight to the forecasted lags that contain inherent error. I decided only to use lags outside of the range of the forecast. The lag of 28 will also be used to create the baseline model, where the prediction will be the amount of units sold 28 days before. I also made a column that gives the average sales of all the lags which are all the same day of the week.

In [27]:
```
# make an iterable array of all the unique ids
items = CA1_F1['id'].unique()
# there are 216 unique items in the dataframe
len(items)
```

Out[27]: 216

In [28]:
```
# period will be 216 items * n days
CA1_F1_price['lag_28'] = CA1_F1_price['sales'].shift(periods=216*28)
CA1_F1_price['lag_35'] = CA1_F1_price['sales'].shift(periods=216*35)
CA1_F1_price['lag_42'] = CA1_F1_price['sales'].shift(periods=216*42)
CA1_F1_price['lag_49'] = CA1_F1_price['sales'].shift(periods=216*49)
```

In [29]:
```
# creates a column
CA1_F1_price['avg'] = (CA1_F1_price['lag_28'] + CA1_F1_price['lag_35'] + CA1_
```

In [30]:
```
# drops observations with missing values
CA1_F1_price.dropna(inplace=True)
```

In [31]:
```
# converts the d variable into an integer
CA1_F1_price['d'] = CA1_F1_price['d'].str.replace('d_', '').astype(int)
```

Baseline Models

A common way to make baseline models for time series analysis is to calculate a lag over a certain period. As the goal is to forecast sales 28 days out, the lag_28 column for the last 28 days of the time series will be used as a baseline prediction. This gives an average RMSE of 2.06 between all the different products. There is also a baseline model that predicts units sold as the average amount of units sold daily for that item. This gives a better average RMSE of 1.71. A deployable model will need to greatly outperform both of these baseline models.

In [32]:
```
# make an empty list of preds and trues to add to
preds = []
trues = []
# loop through the CA1_F1_price df by item
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item]
    # select only the last 28 days
    test = ts['2016-03-28':]
    # actual sales for every
    trues.append(test['sales'])
    # the predictions will be the number of sales 28 days before
    preds.append(test['lag_28'])
```

In [33]:
```
# empty list that will contain all the RMSEs
RMSEs = []
# there will be 216 arrays of preds and trues
for i in range(216):
    RMSEs.append(mean_squared_error(trues[i], preds[i], squared=False))
```

In [34]:
```
# calculate the average RMSE for the baseline model
np.mean(RMSEs)
```

Out[34]: 2.0627187993990534

In [35]:
```
# make a new list of preds
preds = []
# get the average number of units sold for every item
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item]
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    # add a list with the average 28 times
    preds.append([train['sales'].mean()]*28)
```

In [36]:
```
# function to return average RMSE of all 216 models
def get_rmse():
    # make a list of the RMSE for all 216 models
    RMSEs = []
    for i in range(216):
        RMSEs.append(mean_squared_error(trues[i], preds[i], squared=False))
    # return the mean of the RMSEs list
    return np.mean(RMSEs)
```

In [37]:
```
# use function for average RMSE
get_rmse()
```

Out[37]: 1.7120488074490898

First Model

Random Forest Models are often a good place to begin when searching for a good model. They are extremely flexible and often give pretty good results even without hyperparemeter tuning. They also run much faster than many more complex models. In fact they are often quicker than simpler models such as K-Nearest Neighbors. This model ended up seriously overfitting and had an average coefficient of determination below negative one on testing data. It did get an RMSE of 1.83, which beats the first baseline of 2.06, but fails to outperform the second baseline of 1.71. Significant improvement is needed to make this model actually worth implementing. Also, strangely enough, this Random Forest model was actually much slower than the Extreme Gradient Boosted model. The Random Forest algorithm will be left behind as it does not seem to be appropriate for this sort of problem.

In [38]:
```
# make a list of all the columns columns
cols = CA1_F1_price.drop(columns=['id', 'sales']).columns
```

In [39]:
```
# instantiate empty lists
preds = []
train_scores = []
test_scores = []
importances = []
# loop through every item
for item in items:
    # create a time series only including intended item
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    # remove last 28 days from training data
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    # split into X and y
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # instantiate a RandomForestRegressor and fit to training
    rf = RandomForestRegressor(random_state=42).fit(X_train, y_train)
    # add predictions to the list
    preds.append(rf.predict(X_test))
    # .score methods are important for evaluating whether model is overfit
    train_scores.append(rf.score(X_train, y_train))
    test_scores.append(rf.score(X_test, y_test))
    # retrieve importances from model
    importances.append(dict(zip(cols, rf.feature_importances_)))
```

In [40]:
```
# mean R-squared of .87
np.mean(train_scores)
```

Out[40]: 0.8702285775343062

In [41]:
```
# massively overfit
np.mean(test_scores)
```

Out[41]: -0.9795491960207451

In [42]:
```
# use function for average RMSE
get_rmse()
```

Out[42]:
```
1.8335955133316726
```

In [43]:
```
# helper function to return dictionary of importances
def get_imp():
    # loop through every column
    feat_imp = []
    for col in cols:
        # get the mean importance for every column
        nums = []
        for imp in importances:
            nums.append(imp[col])
        feat_imp.append(np.mean(nums))
    # return a dictionary with average importances for every column
    return dict(zip(cols, feat_imp))
```

In [44]:
```
# use function for importances
get_imp()
```

Out[44]:

```
{'d': 0.2839337527263052,
 'wm_yr_wk': 0.09841299182976755,
 'snap_CA': 0.022267399457546657,
 'ChanukahEnd': 0.0018633630042057406,
 'Christmas': 0.0003120854394019062,
 'CincoDeMayo': 0.0006420301881387592,
 'ColumbusDay': 0.0008681355667154263,
 'Easter': 0.001688124002863015,
 'EidAlAdha': 0.0012810098594773255,
 'Eidal-Fitr': 0.0010614525800370634,
 "Father'sday": 0.0009243128545181596,
 'Halloween': 0.002317013231742939,
 'IndependenceDay': 0.0010059111788702313,
 'LaborDay': 0.0014614947527288185,
 'LentStart': 0.0019241985076988025,
 'LentWeek2': 0.0007898917634785533,
 'MartinLutherKingDay': 0.0012374370164390986,
 'MemorialDay': 0.0010447899991037264,
 "Mother'sday": 0.0008945560526338063,
```


XGBoost

Extreme Gradient Boost(XGBoost) is one of the most popular modern machine learning algorithms. It is a common winner of machine learning competitions and has been shown effictive in a vast range of domains. Even using the default arguments of the XGBoost algorithm can often give strong results, and these results can improve greatly when the hyperparameters are tuned. As mentioned earlier, this algorithm, unexpectedly, ran faster than the Random Forest algorithm.

However, the results of this model was even worse than the first model with an average RMSE of 1.97, which was almost as bad as the first baseline model. It was also even more overfit than the first model, so the hyperparameters will need to be heavily tuned to get more value out of this.

In [45]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # fit to XGBRegressor with default arguments
    xgb = XGBRegressor(random_state=42).fit(X_train, y_train)
    preds.append(xgb.predict(X_test))
    train_scores.append(xgb.score(X_train, y_train))
    test_scores.append(xgb.score(X_test, y_test))
    importances.append(dict(zip(cols, xgb.feature_importances_)))
```

In [46]:
```
# mean R-squared on training
np.mean(train_scores)
```

Out[46]: 0.9160606916245376

In [47]:
```
# mean R-squared on testing
np.mean(test_scores)
```

Out[47]: -1.1666935250261343

In [48]:
```
# return average RMSE
get_rmse()
```

Out[48]: 1.9671867585832188

In [49]:
```
# return dict of importances
get_imp()
```

Out[49]:
```
{'d': 0.028999506,
 'wm_yr_wk': 0.0,
 'snap_CA': 0.020771552,
 'ChanukahEnd': 0.011113732,
 'Christmas': 0.011568806,
 'CincoDeMayo': 0.0077653164,
 'ColumbusDay': 0.007399521,
 'Easter': 0.011954689,
 'EidAlAdha': 0.009162884,
 'Eidal-Fitr': 0.010208428,
 "Father'sday": 0.0086688455,
 'Halloween': 0.014120667,
 'IndependenceDay': 0.009725765,
 'LaborDay': 0.01041976,
 'LentStart': 0.011618274,
 'LentWeek2': 0.0064562056,
 'MartinLutherKingDay': 0.008294864,
 'MemorialDay': 0.007927818,
 "Mother'sday": 0.0074857497,
```

XGBoost Model 2

Tweedie loss is supposed to be good for distributions with a high proportion of zeros, and the results suggest that this was a great choice. While this model still needs a lot of improvement, now it is at least almost on par with the better baseline model, with an RMSE of 1.73. It is much less overfit to the training data now, but still needs some more regularization, which will be a good next step.

In [50]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # set objective to tweedie to fit predictions to tweedie distribution
    xgb2 = XGBRegressor(random_state=42, objective='reg:tweedie').fit(X_train
    preds.append(xgb2.predict(X_test))
    train_scores.append(xgb2.score(X_train, y_train))
    test_scores.append(xgb2.score(X_test, y_test))
    importances.append(dict(zip(cols, xgb2.feature_importances_)))
```

In [51]:
```
np.mean(train_scores)
```

Out[51]: 0.8717568262817005

In [52]:
```
np.mean(test_scores)
```

Out[52]: -0.567660937689128

In [53]:
```
# return average RMSE
get_rmse()
```

Out[53]: 1.7303743696414524

In [54]:
```
# return dict of importances
get_imp()
```

Out[54]:
```
{'d': 0.032245345,
 'wm_yr_wk': 0.0,
 'snap_CA': 0.018353257,
 'ChanukahEnd': 0.008309086,
 'Christmas': 0.041004755,
 'CincoDeMayo': 0.00761557,
 'ColumbusDay': 0.007632049,
 'Easter': 0.008010554,
 'EidAlAdha': 0.0071339705,
 'Eidal-Fitr': 0.0067807976,
 "Father'sday": 0.0071282764,
 'Halloween': 0.009651025,
 'IndependenceDay': 0.007888835,
 'LaborDay': 0.008188605,
 'LentStart': 0.008128597,
 'LentWeek2': 0.008507379,
 'MartinLutherKingDay': 0.008708451,
 'MemorialDay': 0.007198242,
 "Mother'sday": 0.006291358,
```

XGBoost Model 3

Regularization really helped the model reduce overfitting. While there was a huge improvement, the model is still very overfit. However, the RMSE has reached 1.54, which is finally outperforming both of the baseline models. The model still has a negative coefficient of determination, which is suggests a lot of improvement can be made.

In [55]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # added l1 and l2 regularization
    xgb3 = XGBRegressor(random_state=42, objective='reg:tweedie', reg_alpha=5
    preds.append(xgb3.predict(X_test))
    train_scores.append(xgb3.score(X_train, y_train))
    test_scores.append(xgb3.score(X_test, y_test))
    importances.append(dict(zip(cols, xgb3.feature_importances_)))
```

In [56]:
```
np.mean(train_scores)
```

Out[56]: 0.6785659786853302

In [57]:
```
np.mean(test_scores)
```

Out[57]: -0.23296383638873552

In [58]:
```
# return average RMSE
get_rmse()
```

Out[58]: 1.544304111249151

In [59]:
```
# return dict of importances
get_imp()
```

Out[59]:
```
{'d': 0.048945446,
 'wm_yr_wk': 0.0,
 'snap_CA': 0.01736595,
 'ChanukahEnd': 0.00917954,
 'Christmas': 0.0445314,
 'CincoDeMayo': 0.0064787115,
 'ColumbusDay': 0.00869104,
 'Easter': 0.008074566,
 'EidAlAdha': 0.007102828,
 'Eidal-Fitr': 0.007302305,
 "Father'sday": 0.0074213357,
 'Halloween': 0.011037641,
 'IndependenceDay': 0.009694539,
 'LaborDay': 0.007946148,
 'LentStart': 0.0098119965,
 'LentWeek2': 0.010197224,
 'MartinLutherKingDay': 0.009844167,
 'MemorialDay': 0.006403624,
 "Mother'sday": 0.0067767864,
```

XGBoost Model 4

The default max_depth for XGBoost is 3, and generally it is recommended to stay below 10. Typically a max depth of 4-8 is the best parameter for XGBoost, as you need the trees to be weak predictors. I tried reducing the max depth in order to reduce overfit, but it ended up causing underfit. Then I tried increasing the max_depth and bumping up regularization to combat overfitting. This seemed to result in overfit that regularization could not deal with. Increasing regularization did have a good effect, however the max_depth of 3 resulted in the best average RMSE. The RMSE was reduced to 1.47, but the model still has a negative coefficient of determination. Much progress will need to be made before this model is worth deploying. There seems to be too much noise to fit these models, or just as likely, there are too many important variables this model is missing.

In [60]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # left max_depth at default of 3 and bumped up l1 and l2 regularization t
    xgb4 = XGBRegressor(random_state=42, objective='reg:tweedie', max_depth=3
    preds.append(xgb4.predict(X_test))
    train_scores.append(xgb4.score(X_train, y_train))
    test_scores.append(xgb4.score(X_test, y_test))
    importances.append(dict(zip(cols, xgb4.feature_importances_)))
```

In [61]:
```
np.mean(train_scores)
```

Out[61]: 0.225347435583728

In [62]:
```
np.mean(test_scores)
```

Out[62]: -0.14892255326148632

In [63]:
```
# return average RMSE
get_rmse()
```

Out[63]: 1.4742527782295138

In [64]:
```
# return dict of importances
get_imp()
```

Out[64]:
```
{'d': 0.11453462,
 'wm_yr_wk': 0.0,
 'snap_CA': 0.012602516,
 'ChanukahEnd': 0.0016057836,
 'Christmas': 0.02925956,
 'CincoDeMayo': 0.0022497613,
 'ColumbusDay': 0.0014691772,
 'Easter': 0.0030411228,
 'EidAlAdha': 0.00084533653,
 'Eidal-Fitr': 0.0017338111,
 "Father'sday": 0.001687361,
 'Halloween': 0.0037634578,
 'IndependenceDay': 0.00334487,
 'LaborDay': 0.0019199215,
 'LentStart': 0.0036812099,
 'LentWeek2': 0.0033802043,
 'MartinLutherKingDay': 0.003780982,
 'MemorialDay': 0.0029031632,
 "Mother'sday": 0.001209296,
```

LGBM Model

Light gradient boosted machine algorithms are designed to work on the same principals as XGB, but work even faster. Many of the top models in the M5-Accuracy competition used the LGBM algorithm, and the faster speed makes it easier to tune hyperparameters for. The default arguments of the LGBM gave much better results than the default XGBoost algorithm, with significantly less overfitting. The default arguments achieved an RMSE of 1.58, which already outperforms baseline. However, more tuning will be needed if the LGBM model is to outperform XGBoost.

In [65]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # default arguments for LGBMRegressor
    lgbm = LGBMRegressor(random_state=42).fit(X_train, y_train)
    preds.append(lgbm.predict(X_test))
    train_scores.append(lgbm.score(X_train, y_train))
    test_scores.append(lgbm.score(X_test, y_test))
    importances.append(dict(zip(cols, lgbm.feature_importances_)))
```

In [66]:
```
np.mean(train_scores)
```

Out[66]:
```
0.6231491889099577
```

In [67]:
```
np.mean(test_scores)
```

Out[67]:
```
-0.3385652572298986
```

In [68]:
```
# return average RMSE
get_rmse()
```

Out[68]:
```
1.5794216034649093
```

In [69]:
```
# return dict of importances
get_imp()
```

Out[69]:
```
{'d': 919.4861111111111,
 'wm_yr_wk': 332.1296296296296,
 'snap_CA': 83.68518518518519,
 'ChanukahEnd': 0.0,
 'Christmas': 0.0,
 'CincoDeMayo': 0.0,
 'ColumbusDay': 0.0,
 'Easter': 0.0,
 'EidAlAdha': 0.0,
 'Eidal-Fitr': 0.0,
 "Father'sday": 0.0,
 'Halloween': 0.0,
 'IndependenceDay': 0.0,
 'LaborDay': 0.0,
 'LentStart': 0.0,
 'LentWeek2': 0.0,
 'MartinLutherKingDay': 0.0,
 'MemorialDay': 0.0,
 "Mother'sday": 0.0,
```

LGBM Model 2

While implementing the tweedie objective for the LGBM model did help, it didn't help nearly as much as the tweedie objective id for the XGBoost algorithm. The RMSE was brought down to 1.57, a slight improvement over the previous model, but it seems that other parameters will need to be tweaked for more improvement.

In [70]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # changed the objective hyperparemeter to tweedie
    lgbm2 = LGBMRegressor(random_state=42, objective='tweedie').fit(X_train,
    preds.append(lgbm2.predict(X_test))
    train_scores.append(lgbm2.score(X_train, y_train))
    test_scores.append(lgbm2.score(X_test, y_test))
    importances.append(dict(zip(cols, lgbm2.feature_importances_)))
```

In [71]:
```
np.mean(train_scores)
```

Out[71]: 0.6920178372604975

In [72]:
```
np.mean(test_scores)
```

Out[72]: -0.2980829367034139

In [73]:
```
# return average RMSE
get_rmse()
```

Out[73]: 1.5667547227755494

In [74]:
```
# return dict of importances
get_imp()
```

Out[74]:
```
{'d': 948.1342592592592,
 'wm_yr_wk': 375.61574074074076,
 'snap_CA': 89.11111111111111,
 'ChanukahEnd': 0.0,
 'Christmas': 0.0,
 'CincoDeMayo': 0.0,
 'ColumbusDay': 0.0,
 'Easter': 0.0,
 'EidAlAdha': 0.0,
 'Eidal-Fitr': 0.0,
 "Father'sday": 0.0,
 'Halloween': 0.0,
 'IndependenceDay': 0.0,
 'LaborDay': 0.0,
 'LentStart': 0.0,
 'LentWeek2': 0.0,
 'MartinLutherKingDay': 0.0,
 'MemorialDay': 0.0,
 "Mother'sday": 0.0,
```

LGBM Model 3

The default arguments for LGBM do not specify max_depth, but instead limits the number of leaves to 31. In order to reduce overfitting, max_depth was set to 3, which would have a maximum of 8 leaves. Three was chosen as it matched the optimal depth of the XGBoost algorithm. This led to a much better RMSE of 1.53, but still needed a good bit of improvement in the overfitting department.

In [75]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # changed max_depth parameter to 3
    lgbm3 = LGBMRegressor(random_state=42, objective='tweedie', max_depth=3).
    preds.append(lgbm3.predict(X_test))
    train_scores.append(lgbm3.score(X_train, y_train))
    test_scores.append(lgbm3.score(X_test, y_test))
    importances.append(dict(zip(cols, lgbm3.feature_importances_)))
```

In [76]:
```
np.mean(train_scores)
```

Out[76]:
```
0.3173486254893772
```

In [77]:
```
np.mean(test_scores)
```

Out[77]:
```
-0.22673440194569652
```

In [78]:
```
# return average RMSE
get_rmse()
```

Out[78]:
```
1.529020861241018
```

In [79]:
```
# return dict of importances
get_imp()
```

Out[79]:
```
{'d': 179.78703703703704,
 'wm_yr_wk': 89.13425925925925,
 'snap_CA': 14.319444444444445,
 'ChanukahEnd': 0.0,
 'Christmas': 0.0,
 'CincoDeMayo': 0.0,
 'ColumbusDay': 0.0,
 'Easter': 0.0,
 'EidAlAdha': 0.0,
 'Eidal-Fitr': 0.0,
 "Father'sday": 0.0,
 'Halloween': 0.0,
 'IndependenceDay': 0.0,
 'LaborDay': 0.0,
 'LentStart': 0.0,
 'LentWeek2': 0.0,
 'MartinLutherKingDay': 0.0,
 'MemorialDay': 0.0,
 "Mother'sday": 0.0,
```

LGBM Model 4

To further reduce overfitting, l1 and l2 regularization was added. This brought the RMSE down to 1.47, on par with the best RMSE of the XGBoost algorithms, albeit with a worse R-squared. Perhaps the depth of 3 is too small since the LGBM models don't overfit as much as the XGBoost models. I'll experiment with that in the next model.

In [80]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # set l1 and l2 regularization to 10
    lgbm4 = LGBMRegressor(random_state=42, objective='tweedie', max_depth=3,
    preds.append(lgbm4.predict(X_test))
    train_scores.append(lgbm4.score(X_train, y_train))
    test_scores.append(lgbm4.score(X_test, y_test))
    importances.append(dict(zip(cols, lgbm4.feature_importances_)))
```

In [81]:
```
np.mean(train_scores)
```

Out[81]: 0.24732014727022897

In [82]:
```
np.mean(test_scores)
```

Out[82]: -0.16713719880589933

In [83]:
```
# return average RMSE
get_rmse()
```

Out[83]: 1.4738177095328013

In [84]:
```
# return dict of importances
get_imp()
```

Out[84]:
```
{'d': 190.91666666666666,
 'wm_yr_wk': 95.9675925925926,
 'snap_CA': 10.069444444444445,
 'ChanukahEnd': 0.0,
 'Christmas': 0.0,
 'CincoDeMayo': 0.0,
 'ColumbusDay': 0.0,
 'Easter': 0.0,
 'EidAlAdha': 0.0,
 'Eidal-Fitr': 0.0,
 "Father'sday": 0.0,
 'Halloween': 0.0,
 'IndependenceDay': 0.0,
 'LaborDay': 0.0,
 'LentStart': 0.0,
 'LentWeek2': 0.0,
 'MartinLutherKingDay': 0.0,
 'MemorialDay': 0.0,
 "Mother'sday": 0.0,
```

LGBM Model 5

I figured that it would be a good idea to capture more complexity in the model here by increasing the size of the trees, but balanced out the complexity by bumping up the regularization to higher levels to parse through the noise. This gave a slightly better RMSE of 1.47, but was not extremely impactful on performance.

In [85]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # changed num_leaves to 64 and max_depth 6, and bumped up regularization
    lgbm5 = LGBMRegressor(random_state=42, objective='tweedie', num_leaves=64
    preds.append(lgbm5.predict(X_test))
    train_scores.append(lgbm5.score(X_train, y_train))
    test_scores.append(lgbm5.score(X_test, y_test))
    importances.append(dict(zip(cols, lgbm5.feature_importances_)))
```

In [86]:
```
np.mean(train_scores)
```

Out[86]: 0.24443125477495545

In [87]:
```
np.mean(test_scores)
```

Out[87]: -0.18778630355894385

In [88]:
```
# return average RMSE
get_rmse()
```

Out[88]: 1.471088475091799

In [89]:
```
# return dict of importances
get_imp()
```

Out[89]:
```
{'d': 374.43981481481484,
 'wm_yr_wk': 210.9212962962963,
 'snap_CA': 8.407407407407407,
 'ChanukahEnd': 0.0,
 'Christmas': 0.0,
 'CincoDeMayo': 0.0,
 'ColumbusDay': 0.0,
 'Easter': 0.0,
 'EidAlAdha': 0.0,
 'Eidal-Fitr': 0.0,
 "Father'sday": 0.0,
 'Halloween': 0.0,
 'IndependenceDay': 0.0,
 'LaborDay': 0.0,
 'LentStart': 0.0,
 'LentWeek2': 0.0,
 'MartinLutherKingDay': 0.0,
 'MemorialDay': 0.0,
 "Mother'sday": 0.0,
```

Final Model

While XGBoost achieved better numbers for the coefficient of determination, the LGBM models had better RMSEs. I decided to choose this LGBM model because it is faster than the XGBoost model and the RMSE is the metric I value more. The final model has similiar parameters to the tuned LGBM model 5, but I did a little bit of tweaking to get a little more accuracy out of it. The final RMSE was 1.46. Despite this being my final model, there is still a lot of improvement that needs to be made before it is of much value. I never got the R-squared positive for the model, which means that I must have been missing some useful patterns by not having the right features. My model included variables that would be able to replicate time series data, such as lags, rolling means, and weekdays, but vastly underperformed models that used N-BEATS, which is made for time series data. This model is a decent start, but is missing factors that could make it competitive with the top models from the competition. I may need to expand my knowledge before I can make a model on par with others.

In [90]:
```
preds = []
train_scores = []
test_scores = []
importances = []
for item in items:
    ts = CA1_F1_price[CA1_F1_price['id'] == item].drop(columns=['id'])
    train = ts[:'2016-03-28']
    test = ts['2016-03-28':]
    X_train = train.drop('sales', axis=1)
    X_test = test.drop('sales', axis=1)
    y_train = train['sales']
    y_test = test['sales']
    # increase the number of estimators to 500, increase the depth of trees,
    lgbm6 = LGBMRegressor(random_state=42, n_estimators=500, objective='tweed
    preds.append(lgbm6.predict(X_test))
    train_scores.append(lgbm6.score(X_train, y_train))
    test_scores.append(lgbm6.score(X_test, y_test))
    importances.append(dict(zip(cols, lgbm6.feature_importances_)))
```

In [91]:
```
np.mean(train_scores)
```

Out[91]: 0.24061940543395086

In [92]:
```
np.mean(test_scores)
```

Out[92]: -0.1855647671614698

In [93]:
```
# return average RMSE
get_rmse()
```

Out[93]: 1.463227842223587

In [94]:
```
# return dict of importances
feat_importance = get_imp()
feat_importance
```

Out[94]:
```
{'d': 503.6388888888889,
 'wm_yr_wk': 318.31944444444446,
 'snap_CA': 9.087962962962964,
 'ChanukahEnd': 0.0,
 'Christmas': 0.0,
 'CincoDeMayo': 0.0,
 'ColumbusDay': 0.0,
 'Easter': 0.0,
 'EidAlAdha': 0.0,
 'Eidal-Fitr': 0.0,
 "Father'sday": 0.0,
 'Halloween': 0.0,
 'IndependenceDay': 0.0,
 'LaborDay': 0.0,
 'LentStart': 0.0,
 'LentWeek2': 0.0,
 'MartinLutherKingDay': 0.0,
 'MemorialDay': 0.0,
 "Mother'sday": 0.0,
 'NBAFinalsEnd': 0.0,
 'NBAFinalsStart': 0.0,
 'NewYear': 0.0,
 'None': 6.037037037037037,
 'OrthodoxChristmas': 0.0,
 'OrthodoxEaster': 0.0,
 'PesachEnd': 0.0,
 'PresidentsDay': 0.0,
 'PurimEnd': 0.0,
 'Ramadanstarts': 0.0,
 'StPatricksDay': 0.0,
 'SuperBowl': 0.0,
 'Thanksgiving': 0.0,
 'ValentinesDay': 0.0,
 'VeteransDay': 0.0,
 'x0_Friday': 11.75462962962963,
 'x0_Monday': 8.708333333333334,
 'x0_Saturday': 27.708333333333332,
 'x0_Sunday': 19.203703703703702,
 'x0_Thursday': 6.708333333333333,
 'x0_Tuesday': 10.430555555555555,
 'x0_Wednesday': 9.310185185185185,
 'x1_1': 11.125,
 'x1_2': 11.837962962962964,
 'x1_3': 8.208333333333334,
 'x1_4': 6.481481481481482,
 'x1_5': 6.925925925925926,
 'x1_6': 12.800925925925926,
 'x1_7': 15.851851851851851,
 'x1_8': 9.597222222222221,
 'x1_9': 12.708333333333334,
 'x1_10': 12.36574074074074,
 'x1_11': 16.12037037037037,
```

```
'x1_12': 14.523148148148149,
'sell_price': 9.930555555555555,
'lag_28': 46.361111111111114,
'lag_35': 43.300925925925924,
'lag_42': 51.333333333333336,
'lag_49': 45.0462962962963,
'avg': 87.31481481481481}
```

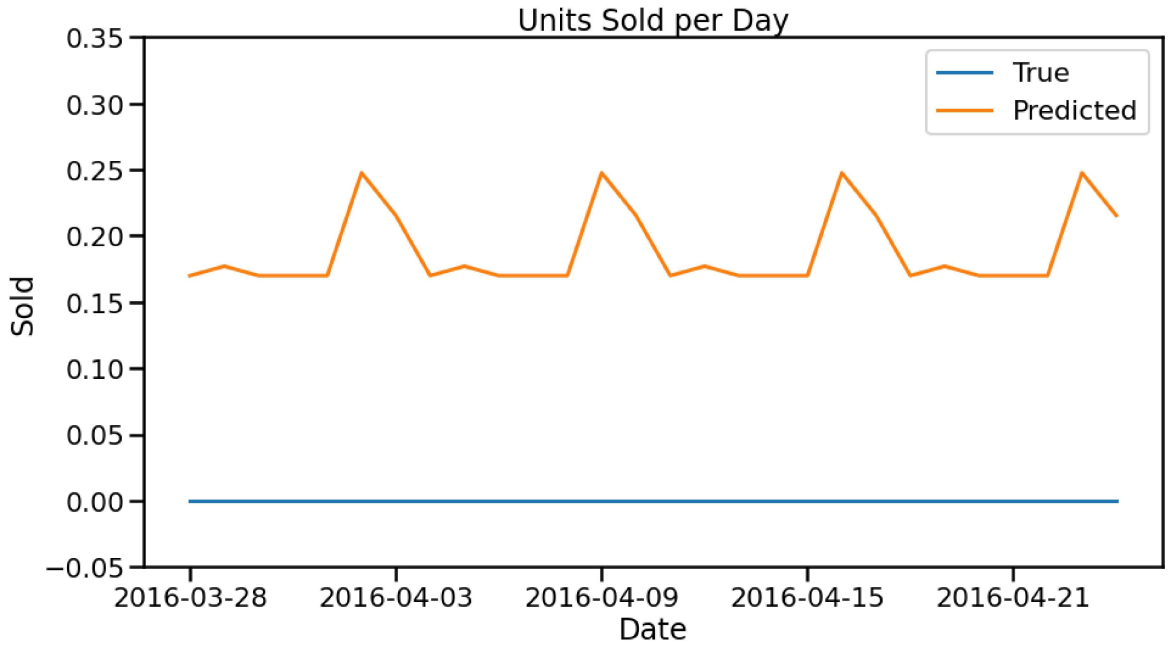
Final Model Visualizations

While the model didn't perform much better than baseline, it still showed a few promising patterns. You can see an obvious weekly pattern emerge from the graph of items 7's prediction. The item was sold at a low frequency, so there was never a full item predicted to be sold, but sometimes an item is sold. While this model would never predict a unit to be sold for any individual day, it would predict a unit to be sold once every 4 or 5 days. This isn't an ideal replication, but it is getting at representing the infrequency of units sold for this item, which is worth something.

In [95]:
```
# set the seaborn context to poster to make graphs better for presentations
sns.set_context('poster')
```

In [97]:
```
# make the 7th prediction array into a series with a datetime index
pred = pd.Series(preds[7], index=trues[7].index)
```

In [98]:
```
# visualize the models performance on an item with no sales
pred = pd.Series(preds[7], index=trues[7].index)
fig, ax = plt.subplots(figsize=(15, 8))
sns.lineplot(data=trues[7], ax=ax)
sns.lineplot(data=pred, ax=ax)
ax.set_xticks(['2016-03-28', '2016-04-03', '2016-04-09', '2016-04-15', '2016-
ax.set_ybound(.35)
ax.set_ylim(-.05)
ax.set_ylabel('Sold')
ax.set_xlabel('Date')
ax.set_title('Units Sold per Day')
ax.legend(['True', 'Predicted'])
plt.savefig('../images/no_units_forecast.jpeg', facecolor='white');
```



I also graphed how the model fared for predicting the highest selling product in the department. It ended up always predicting much closer to the average number of sales and never predicting as low for the low volumne days or as high as the high volume days. This is definitely expected given just how much of sales are up to random chance, but models from others seemed to do a better job catching more of this randomness after looking through some of the kaggle results. You see this pattern pretty clearly in the graph the Total Units Sold per Day graph below as well. It shows the total number of items sold in the department every day. However, the model did do much better when predicting total sales for the items across the entire 28 days. For example, the model predicted 288 units sold for FOODS1 Item 218 across the 28 days. This was only 47 more than the 241 that were actually sold in that period, and this was for the highest selling item. When I compared the predicted total units sold with the actual units sold across the 28 days, the RMSE was only 23.1 units. When you spread this RMSE of 23.1 units across 28 days, the results seem a little more acceptable.

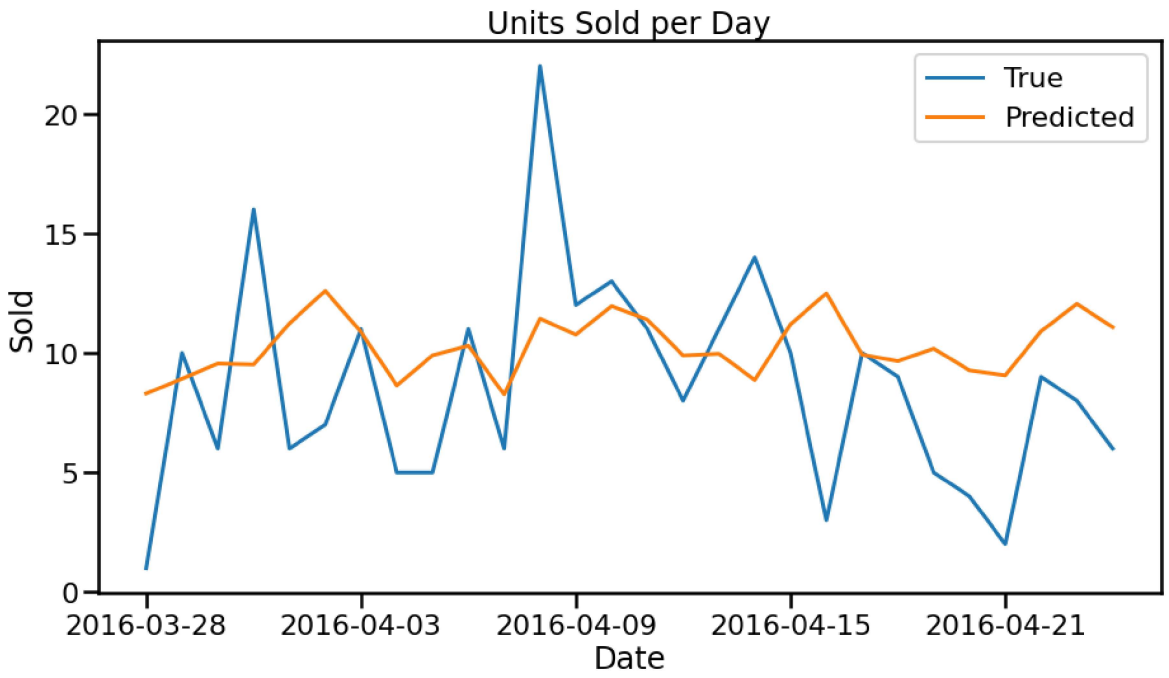
In [99]:
```
# Find the item with the most sales
CA1_F1_price.groupby('id')['sales'].sum().sort_values().tail(1)
```

Out[99]:
```
id
FOODS_1_218_CA_1_validation    22196
Name: sales, dtype: int64
```

In [100]:
```
# print the index of the item above
for i, item in enumerate(items):
    if item == 'FOODS_1_218_CA_1_validation':
        print(i)
```

```
214
```

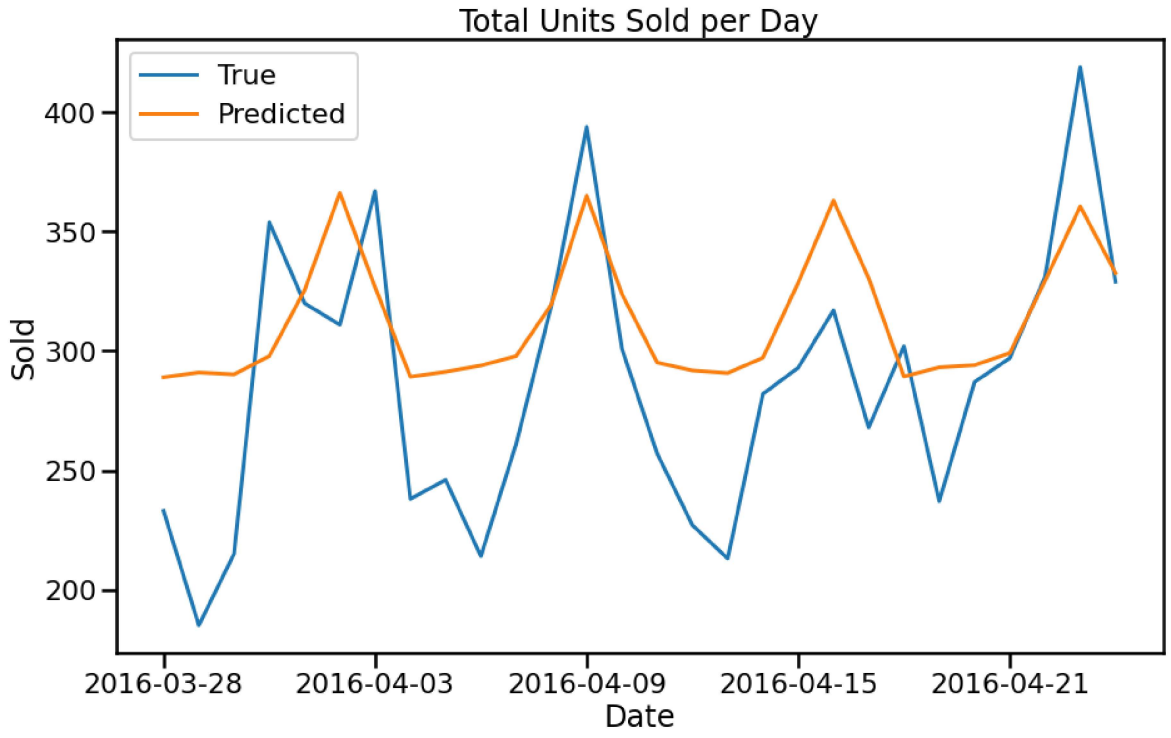
In [101]:
```
# visualize the models performance on the item with the most sales
pred = pd.Series(preds[214], index=trues[214].index)
fig, ax = plt.subplots(figsize=(15, 8))
sns.lineplot(data=trues[214], ax=ax)
sns.lineplot(data=pred, ax=ax)
ax.legend(['True', 'Predicted'])
ax.set_ylabel('Sold')
ax.set_xlabel('Date')
ax.set_xticks(['2016-03-28', '2016-04-03', '2016-04-09', '2016-04-15', '2016-
ax.set_title('Units Sold per Day')
plt.savefig('../images/many_units_forecast.jpeg', facecolor='white');
```



In [102]:
```
# list of total number of items predicted to sell and actually sold
monthly_preds = []
monthly_trues = []
for i in range(28):
    daily_preds = []
    for pred in preds:
        daily_preds.append(pred[i])
    monthly_preds.append(sum(daily_preds))
    daily_trues = []
    for true in trues:
        daily_trues.append(true[i])
    monthly_trues.append(sum(daily_trues))
```

In [103]:
```
# add datetime index to the lists made above
monthly_trues = pd.Series(monthly_trues, index=trues[0].index)
monthly_preds = pd.Series(monthly_preds, index=trues[0].index)
```

In [104]:
```
# visualize how the model predicted total units sold per day
fig, ax = plt.subplots(figsize=(15,9))
sns.lineplot(data=monthly_trues, ax=ax)
sns.lineplot(data=monthly_preds, ax=ax)
ax.legend(['True', 'Predicted'])
ax.legend(['True', 'Predicted'])
ax.set_ylabel('Sold')
ax.set_xlabel('Date')
ax.set_xticks(['2016-03-28', '2016-04-03', '2016-04-09', '2016-04-15', '2016-
ax.set_title('Total Units Sold per Day')
plt.savefig('../images/total_units_sold_forecast.jpeg', facecolor='white');
```



In [105]:
```
# 28 day total of units predicted to be sold for every item
pred_totals = []
for pred in preds:
    pred_totals.append(sum(pred))
```

In [106]:
```
# 28 day total of units actually sold for every item
true_totals = []
for true in trues:
    true_totals.append(sum(true))
```

In [107]:
```
# return root mean squared error between these totals
mean_squared_error(true_totals, pred_totals, squared=False)
```

Out[107]: 23.084728057512034

In [108]:
```
# total predicted units sold for highest volume item
sum(preds[214])
```

Out[108]: 288.1871987832873

In [109]:
```
# actual units sold for highest volume item
sum(trues[214])
```

Out[109]: 241

Most Important Features

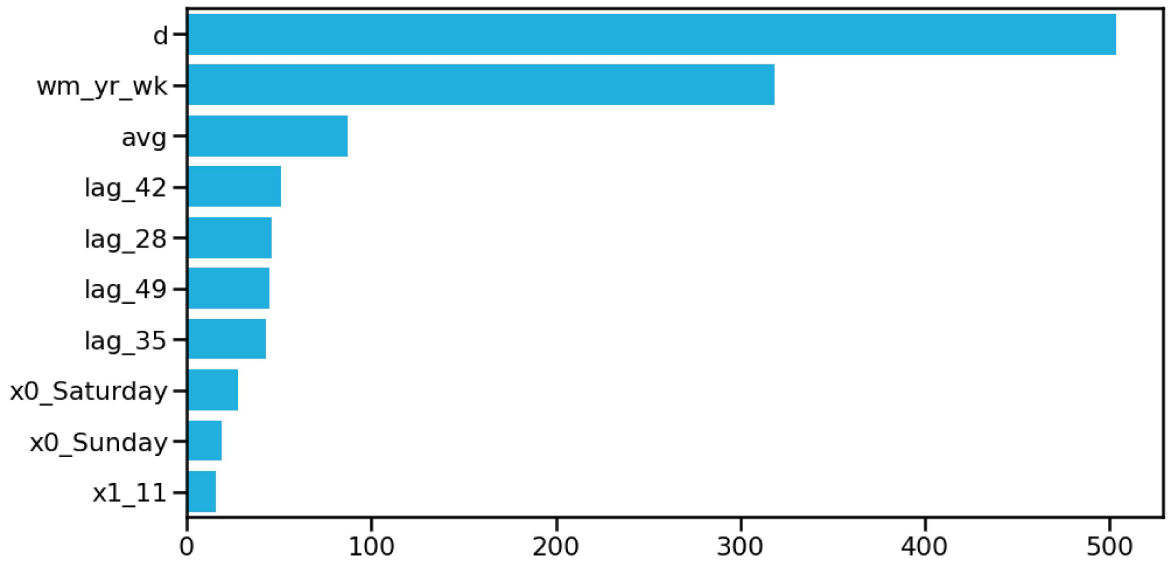
Here, we see that day and week were the most influential factors in predicting the number of sales. This suggests that it was picking up on general long trends, which is good. The third most important feature was the average of the 4 lags, which I also thought would be somewhat analagous to a moving average. The lags were the next most important features, but it makes sense that avg was more useful for the model. The next most important features were Saturday and Sunday, which makes sense, as a lot more people are free on the weekends and buying more products. Lastly, to round out the top ten features is the month November. Since this is a food department, perhaps this captured food items that are sold in high quantities for holidays, especially Thanksgiving.

In [110]:
```
imp_graph = dict(sorted(feat_importance.items(), key=lambda item: item[1], re
```

In [111]:
```
sns.set_palette(sns.color_palette("light:b"))
```

In [112]:
```
fig, ax = plt.subplots(figsize=(15,8))
sns.barplot(x=list(imp_graph.values()), y=list(imp_graph.keys()), color='deep
```

Out[112]: <AxesSubplot:>



Conclusion

While a large amount of progress was made during the modeling process, the final results still leave much to be desired. Despite my best efforts, I never managed to find a model with a positive R-squared or an RMSE below 1.4. In order to make progress I'll need some more time and exposure to time series modeling techniques. While this model didn't perform as well as I wanted it to, this isn't completely unexpected with the level of noise found in the data. Given the circumstances, I'm glad I was able to make a good amount of progress throughout this project and learn as much as I did.

Recommendations

While this current model is not good enough to be deployable, I did think about a few implementations that could be used with a more accurate model.

1. The most obvious implementation of this model would be to predict the number of sales for efficient management of inventory. This could allow stores to know where to send product from distribution centers, how much to send, and when. This would allow for a less cluttered, more efficient store
2. The model could also be used to flag products that show patterns of increasing or decreasing demand. If some product was being sold in significantly higher or lower quantities than the

model expected, that product could be flagged and the number of units ordered monthly could be altered appropriately.
3. The model could also be used to detect how different marketing tactics effect sales. For example, if an item started selling twice as many units as predicted when moved from a bottom shelf to an eye-level shelf, you could make several interesting observations. You could give an estimated value of that shelf space and sell it. You could also analyze how shelf location compares to Television ads or other marketing initiatives.

Next Steps

I had several thoughts on how I could turn this project into an actually deployable model. There are several different approaches that I believe could give my model the information it needs to succeed.

1. One pattern I saw when skimming through models that performed better is that they used models with more explicit time-series components. Several used the deep learning technique, N-BEATS, to capture the time-series components of the model. Other successful models used Amazon's popular algorithm, DeepAR. While I included some time-series related features, my model was unable to perform as well as these models.
2. Another feature that I think would've been useful is having something to estimate the salience of the object in the store. If there were promotions or advertisements, they would undoubtedly make people more likely to buy one of the prodcuts depicted. Also, Items that are closer to eye-level, or in prominent places like end-caps, would also make people more likely to purchase the product. Having a feature that could capture some of these components of visibility could greatly improve the model.
3. Another weakness of the model is that there was no way to capture inventory. While it would be impossible to have an exact number for inventory when forecasting 28 days out, having some way to include inventory in the model would undoubtedly help. Furthermore, if you run out of inventory and sell zero units, you are no longer actually representing demand. This model likely underestimates demand, as it is impossible to tell when there are no units sold due to lack of inventory on the shelf or lack of demand.