Identifying distribution of supply/demand for Walmart products on statewise data

The holiday season is coming, meaning additional requirement of supermarkets wharehouses. On top of this there is a global supply chain crisis going on as a direct result of Covid and the blocking of the Suez Canal. This calls for new tools in the supply chain management.

This notebook develops such a tool for the supermarket chain Walmart using the Kaggle competition dataset "Estimate the unit sales of Walmart retail goods (https://www.kaggle.com/c/m5-forecasting-accuracy/data?select=sell_prices.csv)" on Walmart products quantity sold as well as salesprice on dates starting in 2011 to 2016. This could of course be improved by Walmart supplying their own data.

While Walmart has already done some conventional implementations of hiring additional workers, drivers, trucks etc. (https://corporate.walmart.com/newsroom/2021/10/08/how-walmart-is-navigating-the-supply-chain-to-deliver-this-holiday-season (https://corporate.walmart.com/newsroom/2021/10/08/how-walmart-is-navigating-the-supply-chain-to-deliver-this-holiday-season)) This notebook builds a model that enables estimations of product quantity requirements, meaning that the purchase agents can ensure that Walmart has enough of each product, but not wasting warehouse space on unnecessary quantity of products.

The goal of this assignment will be to solve the issue of building a model that can be used for distribution warehouses to predict the sales pr product pr day. The purpose of this is to give the central distribution warehouses a chance to stock up on these products so the stores always have the products the customer seeks.

To build this model, we've gone through several sections of devolpment:

- Firstly, we import and preprocess the data to make it ready for analysis and perform EDA to examine the available data using different plots, and visually inspecting patterns of interest in the data.
- Secondly, we build two supervised machine learning models, to see if these can accurately predict the quantity requirements.
- Thirdly we use a neural network to build a model that attempts to accurately predicts the quantity requirements, based on the variables presented in our kaggle data.

Functioning notebook link:
https://colab.research.google.com/drive/1H6aJ0aSGR8RxiOOnN0OYOAl1IDnewuGT?usp=sharing (https://colab.research.google.com/drive/1H6aJ0aSGR8RxiOOnN0OYOAl1IDnewuGT?usp=sharing)

In [ ]:

```
#Importing the relevant packes that gets used throughout
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import numpy
import math
from math import sqrt

from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.preprocessing.sequence import TimeseriesGenerator

import tensorflow as tf
import plotly.graph_objects as go

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
```

In [ ]:

```
# Cloning the github containing the uploadet data
!git clone https://github.com/Kris492b/data_wallmart
```

```
Cloning into 'data_wallmart'...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 10 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (10/10), done.
```

In [ ]:

```
# Using unrar to unpackage the uploadet data.
!unrar x "/content/data_wallmart/calendar.rar" "/content/data_wallmart"
!unrar x "/content/data_wallmart/sell_prices.rar" "/content/data_wallmart"
!unrar x "/content/data_wallmart/sales_train_evaluation.rar" "/content/data_wallmart"
# Loading the 3 seperate datasets
data_calendar = pd.read_csv('/content/data_wallmart/calendar.csv')
data_sell_prices = pd.read_csv('/content/data_wallmart/sell_prices.csv')
data_sales = pd.read_csv('/content/data_wallmart/sales_train_evaluation.csv')
```

```
UNRAR 5.50 freeware      Copyright (c) 1993-2017 Alexander Roshal


Extracting from /content/data_wallmart/calendar.rar

Extracting  /content/data_wallmart/calendar.csv
     99%       OK
All OK

UNRAR 5.50 freeware      Copyright (c) 1993-2017 Alexander Roshal


Extracting from /content/data_wallmart/sell_prices.rar

Extracting  /content/data_wallmart/sell_prices.csv
     42%     84%     99%       OK
All OK

UNRAR 5.50 freeware      Copyright (c) 1993-2017 Alexander Roshal


Extracting from /content/data_wallmart/sales_train_evaluation.rar

Extracting  /content/data_wallmart/sales_train_evaluation.csv
     26%     53%     80%     99%       OK
All OK
```

EDA

Now that we have loaded the data for analysis, we decided to make some initial exploratory data analysis to get a quick look at the data we are working with, since the scale makes its hard to grasp while looking at the dataframes.

NOTE: Plots are only shown in the functioning notebook

In [ ]:

```
#Creating a loop for the possibility to merge the data
d_col = [c for c in data_sales.columns if 'd_' in c]
#We are creating a new dataframe where it is based on the data_calender and data_sales
 where we are using the d_col to transpose the id where it creates the posibility to me
rge the two dataframe together
df_for_specific_stores_CA = pd.merge(data_calendar.set_index('d'),data_sales.set_index(
'id')[d_col].T, left_index=True, right_index=True)
#We are resetting the index where we set the index as "date" where we want rename the c
olumn from "index" to "d".
df_for_specific_stores_CA = df_for_specific_stores_CA.reset_index().set_index('date').r
ename(columns={'index':'d'})
```

In [ ]:

```
#Creating a list of all stores available in our dataset
column_list= data_sales['store_id'].unique()
column_list
```

Out[ ]:

```
array(['CA_1', 'CA_2', 'CA_3', 'CA_4', 'TX_1', 'TX_2', 'TX_3', 'WI_1',
       'WI_2', 'WI_3'], dtype=object)
```

In [ ]:

```
#Because of the size of the data, we choose to only look at the stores in California, a
s an example
the_list_of_stores_in_CA =['CA_1', 'CA_2', 'CA_3', 'CA_4']
fig = go.Figure()
# We are creating a loop to go through the "the_list_of_stores_in_CA" where it goes fro
m CA_1-CA_4 by having the loop
for c in the_list_of_stores_in_CA:
  # We are creating another loop where we look at df_for_specific_stores_CA and look at
the column where we define which rows it needs to include for the figure by using the i
f statement.
    store_items = [w for w in df_for_specific_stores_CA.columns if c in w]
  #Creating the results we want to investigate and using the rolling to get a rolling m
ean based on 7 days
    data_results_stores_total = df_for_specific_stores_CA[store_items].sum(axis=1).roll
ing(7).mean()
   #We feed the model with the above code.
    fig.add_trace(go.Scatter(x=data_results_stores_total.index, y=data_results_stores_t
otal, name=c))
    #Updating the layout for the figure with the title etc.
fig.update_layout(yaxis_title="Sales for all categories", xaxis_title="Year distributio
n", title="Displaying the rolling average for 7 daily sale(for each store in Californi
a)")
```

The figure displays the distribution of every store in California based the total amount of "cat_items" which is "hobbies, household, foods". The figure shows a peak and downfall between the months. Across the different stores it seems like all of them follow the same distribution and the amount of sold units increases slowly. CA_3 is the store which have been selling more units than the other stores.

The distribution of the different type of product categories based on California

In [ ]:

```
#Doing the same procedure as above where we create a new dataframe where we remove the
 "state_id" where the rows in only contain "CA".
df_that_contain_only_CA_stores=data_sales[data_sales["state_id"].str.contains("CA")==Tr
ue]
```

In [ ]:

```
#Same as above where we want to create the possibility to merge the two data frames.
d_col = [c for c in df_that_contain_only_CA_stores.columns if 'd_' in c]

df_based_on_specific_cate_CA = pd.merge(data_calendar.set_index('d'),df_that_contain_on
ly_CA_stores.set_index('id')[d_col].T, left_index=True, right_index=True)

df_based_on_specific_cate_CA = df_based_on_specific_cate_CA.reset_index().set_index('da
te').rename(columns={'index':'d'})
```

In [ ]:

```
cat_list = data_sales.cat_id.unique()
fig = go.Figure()
for k in cat_list:
    cat_items = [x for x in df_based_on_specific_cate_CA.columns if k in x]
    data_results_cat= df_based_on_specific_cate_CA[cat_items].sum(axis=1).rolling(7).me
an()
    fig.add_trace(go.Scatter(x=data_results_cat.index, y=data_results_cat, name=k))
fig.update_layout(yaxis_title="Amount of sales", xaxis_title="Year distribution", title
="Rolling Average 7 Sales against time (per category) based on the stores in Californi
a")
```

As seen above the graph shows the distribution of sold units through the years where we have summarized the different categories based on all the stores in Califoria which is 4 stores. The graph shows that there is a peak in the middle of the years of foods which could be explained by events etc, but it can be hard to decipher when we dont know what the product is. The same characteristic can be seen in household where there also is a peak in the middle of the year. Hobbies on the other hand do not really increase in the amount of sold units through the years and there is not really a peak compared to the others.

So at the end of this EDA we realise we are dealing with a vast datasat which can be modified in alot of ways to fit the need of walmart with regards to our problem statement. With that being said though, since we have had several issues with memory during the preproccesssing we have chosen to go further with a single product, which we want to track across all the stores to get a baseline understanding of how compute the model, which we can then expand with the rest of the products when / if needed.

Preprocessing data

This part of the notebook will contain the preprocessing actions we have decided to use on the dataset before starting any machine learning or neural network modelling

In [ ]:

```
# Previewing and taking a look at the calendar data
# The more obscure columns are the following:
# 'wm_yr_wk', which is a week number from the first observation, counting up in instanc
es of 52
# 'd' is the amount of days of recorded observations, counting up.
# 'event_name' and 'event_type' is holidays, religious and or sporting event days throu
ghout the calendar year, coded for name and type of event.
# The reason there is two event_type/name is that some events may overlap, therefore mu
ltiple events may happen on the same date.
# 'Snap_x' is the ability to use certain applicable discount coupons in the Wallmart st
ores in the relative state
data_calendar
```

Out[ ]:

	date	wm_yr_wk	weekday	wday	month	year	d	event_name_1	event_type_
0	2011-01-29	11101	Saturday	1	1	2011	d_1	NaN	Na
1	2011-01-30	11101	Sunday	2	1	2011	d_2	NaN	Na
2	2011-01-31	11101	Monday	3	1	2011	d_3	NaN	Na
3	2011-02-01	11101	Tuesday	4	2	2011	d_4	NaN	Na
4	2011-02-02	11101	Wednesday	5	2	2011	d_5	NaN	Na
...	...	...	...	...	...	...	...	...	
1964	2016-06-15	11620	Wednesday	5	6	2016	d_1965	NaN	Na
1965	2016-06-16	11620	Thursday	6	6	2016	d_1966	NaN	Na
1966	2016-06-17	11620	Friday	7	6	2016	d_1967	NaN	Na
1967	2016-06-18	11621	Saturday	1	6	2016	d_1968	NaN	Na
1968	2016-06-19	11621	Sunday	2	6	2016	d_1969	NBAFinalsEnd	Sporti

1969 rows × 14 columns

In [ ]:

```
# Previewing the sell prices for the 3 product catagories, Hobbie, Household and Food f
or any given store id.
# Note that the sell price of products only updates pr. week. This caveat will be expan
ded upon later.
data_sell_prices
```

Out[ ]:

	store_id	item_id	wm_yr_wk	sell_price
0	CA_1	HOBBIES_1_001	11325	9.58
1	CA_1	HOBBIES_1_001	11326	9.58
2	CA_1	HOBBIES_1_001	11327	8.26
3	CA_1	HOBBIES_1_001	11328	8.26
4	CA_1	HOBBIES_1_001	11329	8.26
...	...	...	...	...
6841116	WI_3	FOODS_3_827	11617	1.00
6841117	WI_3	FOODS_3_827	11618	1.00
6841118	WI_3	FOODS_3_827	11619	1.00
6841119	WI_3	FOODS_3_827	11620	1.00
6841120	WI_3	FOODS_3_827	11621	1.00

6841121 rows × 4 columns

As it can be seen in data_sell_prices, there is 6.800.000+ columns of prices, and we have found that the amount of ram supplied by Google colab was insufficient at handling the amount of data in the sets, which is why we want to reduce the data amount in the following section

In [ ]:

```
# Reducing the product amount, thereby drastically reducing the data amount and improvi
ng runtime for colab.
discard_items = ["HOUSEHOLD", 'HOBBIES']
# We define a list of items to discard from the data, and find observations in the item
_id column where the string contains the search woods from the list above.
data_sell_prices = data_sell_prices[~data_sell_prices.item_id.str.contains('|'.join(dis
card_items))]



#As stated above, we can remove all observations not within the state of our choice, by
using same function as above.
#discard_state = ["WI", "TX"]
#data_sell_prices = data_sell_prices[~data_sell_prices.store_id.str.contains('|'.join(d
iscard_state))]

# If we choose to subset like this, we can also remove the following variables to fit t
he data
#data_calendar.drop(columns=['snap_TX','snap_WI', "snap_CA"], inplace=True)
```

In [ ]:

```
# Displaying the updated sell price dataframe with around a half of the observations. T
he rest will be subset by our merges, but this amount makes it easier to work with.
data_sell_prices
```

Out[ ]:

	store_id	item_id	wm_yr_wk	sell_price
368746	CA_1	FOODS_1_001	11101	2.0
368747	CA_1	FOODS_1_001	11102	2.0
368748	CA_1	FOODS_1_001	11103	2.0
368749	CA_1	FOODS_1_001	11104	2.0
368750	CA_1	FOODS_1_001	11105	2.0
...	...	...	...	...
6841116	WI_3	FOODS_3_827	11617	1.0
6841117	WI_3	FOODS_3_827	11618	1.0
6841118	WI_3	FOODS_3_827	11619	1.0
6841119	WI_3	FOODS_3_827	11620	1.0
6841120	WI_3	FOODS_3_827	11621	1.0

3181789 rows × 4 columns

In [ ]:

```
# Displaying the last dataset, the sales amount distributed upon the days of observatio
ns for the products, based on the specific store and product.
data_sales
```

Out[ ]:

	id	item_id	dept_id	cat_id	store_id	sta
0	HOBBIES_1_001_CA_1_evaluation	HOBBIES_1_001	HOBBIES_1	HOBBIES	CA_1	
1	HOBBIES_1_002_CA_1_evaluation	HOBBIES_1_002	HOBBIES_1	HOBBIES	CA_1	
2	HOBBIES_1_003_CA_1_evaluation	HOBBIES_1_003	HOBBIES_1	HOBBIES	CA_1	
3	HOBBIES_1_004_CA_1_evaluation	HOBBIES_1_004	HOBBIES_1	HOBBIES	CA_1	
4	HOBBIES_1_005_CA_1_evaluation	HOBBIES_1_005	HOBBIES_1	HOBBIES	CA_1	
...	...	...	...	...	...	
30485	FOODS_3_823_WI_3_evaluation	FOODS_3_823	FOODS_3	FOODS	WI_3	
30486	FOODS_3_824_WI_3_evaluation	FOODS_3_824	FOODS_3	FOODS	WI_3	
30487	FOODS_3_825_WI_3_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_3	
30488	FOODS_3_826_WI_3_evaluation	FOODS_3_826	FOODS_3	FOODS	WI_3	
30489	FOODS_3_827_WI_3_evaluation	FOODS_3_827	FOODS_3	FOODS	WI_3	

30490 rows × 1947 columns

In [ ]:

```
#Here we can choose any product name to focus on, which can further help our memory iss
ues, in this case we choose a product at random
data_sales = data_sales[data_sales['item_id'].str.contains("FOODS_3_825")]
#Here we can choose to subset to any particulear state, if we so want. This could also
 work with "store_id" if we would rather focus on 1 store over 1 state.
#data_sales = data_sales[data_sales['state_id'].str.contains("CA")]
data_sales
```

Out[ ]:

	id	item_id	dept_id	cat_id	store_id	state_id
3046	FOODS_3_825_CA_1_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_1	CA
6095	FOODS_3_825_CA_2_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_2	CA
9144	FOODS_3_825_CA_3_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_3	CA
12193	FOODS_3_825_CA_4_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_4	CA
15242	FOODS_3_825_TX_1_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_1	TX
18291	FOODS_3_825_TX_2_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_2	TX
21340	FOODS_3_825_TX_3_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_3	TX
24389	FOODS_3_825_WI_1_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_1	WI
27438	FOODS_3_825_WI_2_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_2	WI
30487	FOODS_3_825_WI_3_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_3	WI

10 rows × 1947 columns

In [ ]:

```
# Unpivoting the sales amount to a day column 'd', displaying the amount of sales of th
e given product in a new column 'sales_amount'.
data_sales_product = data_sales.melt(['id','item_id','dept_id','cat_id','store_id','sta
te_id'], var_name='d', value_name='sales_amount')
data_sales_product
```

Out[ ]:

	id	item_id	dept_id	cat_id	store_id	state_id
0	FOODS_3_825_CA_1_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_1	CA
1	FOODS_3_825_CA_2_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_2	CA
2	FOODS_3_825_CA_3_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_3	CA
3	FOODS_3_825_CA_4_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_4	CA
4	FOODS_3_825_TX_1_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_1	TX
...	...	...	...	...	...	...
19405	FOODS_3_825_TX_2_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_2	TX
19406	FOODS_3_825_TX_3_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_3	TX
19407	FOODS_3_825_WI_1_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_1	WI
19408	FOODS_3_825_WI_2_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_2	WI
19409	FOODS_3_825_WI_3_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_3	WI

19410 rows × 8 columns

In [ ]:

```
# Merging the calendar and sales on the day ('d') on data_sales, with the 'inner' metho
d, as to prevent na values on days where there are no observations.
data_walmart = pd.merge(data_sales_product, data_calendar, how='inner', on='d')
```

In [ ]:

```
# A look at the new merged dataframe
data_walmart
```

Out[ ]:

	id	item_id	dept_id	cat_id	store_id	state_id
0	FOODS_3_825_CA_1_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_1	CA
1	FOODS_3_825_CA_2_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_2	CA
2	FOODS_3_825_CA_3_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_3	CA
3	FOODS_3_825_CA_4_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_4	CA
4	FOODS_3_825_TX_1_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_1	TX
...	...	...	...	...	...	...
19405	FOODS_3_825_TX_2_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_2	TX
19406	FOODS_3_825_TX_3_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_3	TX
19407	FOODS_3_825_WI_1_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_1	WI
19408	FOODS_3_825_WI_2_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_2	WI
19409	FOODS_3_825_WI_3_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_3	WI

19410 rows × 21 columns

In [ ]:

```
# Merging the wallmart dataset with the selling prices for the products over time.
# We merge by 3 keys, 'item_id', 'wm_yr_wk' and 'store_id' to make sure that we have th
e correct merging of column values on the three columns.
# Thereby, making sure the item id is correctly paired with selling price, for the corr
ect store id and on the right week number.
# Thereby completing the merging of all datasets into a complete format.
data_walmart = pd.merge(data_walmart, data_sell_prices, how='left', on=['item_id', 'wm_
yr_wk', 'store_id'])
```

In [ ]:

```
# Removing the rows where we do not have a sell price of the product on the respective
 day.
data_walmart = data_walmart[data_walmart['sell_price'].notna()]
```

In [ ]:

```
# Extending the dataset by creating dummy variables for the events and weekdays.
# By creating dummy variables, we gain the advantage of being able to input these event
s and days as features in our neural network.
# We can thereby assume that we gain the correlation of units sold, based upon event, e
vent type and or weekday data.
data_walmart = pd.get_dummies(data=data_walmart, columns=['event_name_1',       'event_
type_1',        'event_name_2', 'event_type_2', 'weekday'])
# Displaying the full dataset with the dummies
data_walmart
```

Out[ ]:

	id	item_id	dept_id	cat_id	store_id	state_id
1	FOODS_3_825_CA_2_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_2	CA
3	FOODS_3_825_CA_4_evaluation	FOODS_3_825	FOODS_3	FOODS	CA_4	CA
4	FOODS_3_825_TX_1_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_1	TX
5	FOODS_3_825_TX_2_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_2	TX
6	FOODS_3_825_TX_3_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_3	TX
...	...	...	...	...	...	...
19405	FOODS_3_825_TX_2_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_2	TX
19406	FOODS_3_825_TX_3_evaluation	FOODS_3_825	FOODS_3	FOODS	TX_3	TX
19407	FOODS_3_825_WI_1_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_1	WI
19408	FOODS_3_825_WI_2_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_2	WI
19409	FOODS_3_825_WI_3_evaluation	FOODS_3_825	FOODS_3	FOODS	WI_3	WI

19389 rows × 64 columns

In [ ]:

```
# In case we would have multiple products, we would keep the id of the product.
#At the same time we could keep the from "store_id" and so on if we decide this ID is i
mportant for distinguishing products
data_walmart = data_walmart.drop(columns=['id','item_id','dept_id','cat_id','store_id',
'state_id', 'wm_yr_wk', 'wday', 'month','year'], axis=1)
```

In [ ]:

```
# Creating a copy of the dataset to manipulate
Walmart_dummy = data_walmart
```

In [ ]:

```
# Aggregating the days in the data. Creating a sum and mean of sales amount and price r
espectively.
# This creates an overall sell price mean and sales amount sum of all Walmart stores in
California over the observed days.
data_walmart_testing = Walmart_dummy.groupby("d", as_index=False).agg({"sales_amount":
"sum",
                                                                       "sell_price":
"mean",
                                                                       })
```

In [ ]:

```
# Removing the already aggregated data, to allow us to aggregate the rest of the data.
Walmart_dummy_two = Walmart_dummy.drop(labels = ["sales_amount", "sell_price", "date"],
axis=1)
```

In [ ]:

```
# Aggregating the remaining data based on observation day.
data_walmart_testing_two = Walmart_dummy_two.groupby("d", as_index=False).agg("mean")
```

In [ ]:

```
# To reinsert the dates in upcoming dataframes, we used the date column, which is corre
ctly indexed, in the previous dataframe and drop the sales amount.
Walmart_dates = Walmart_dummy.iloc[:,:3]
Walmart_dates.drop("sales_amount", axis=1, inplace=True)
# Because this gives duplicate dates, we drop those.
Walmart_dates.drop_duplicates(inplace=True)
# This gives us an observation day index as well as the correct date attached, which ca
n be merged back into the dataframe.
Walmart_dates
```

Out[ ]:

	d	date
1	d_1	2011-01-29
11	d_2	2011-01-30
21	d_3	2011-01-31
31	d_4	2011-02-01
41	d_5	2011-02-02
...	...	...
19360	d_1937	2016-05-18
19370	d_1938	2016-05-19
19380	d_1939	2016-05-20
19390	d_1940	2016-05-21
19400	d_1941	2016-05-22

1941 rows × 2 columns

In [ ]:

```
# Merging all dataframes, to get the daily aggregated data across all stores in CA, sor
ted for date.
Data_walmart_daily = pd.merge(data_walmart_testing, Walmart_dates, how="right", on="d")
Data_walmart_daily = pd.merge(Data_walmart_daily, data_walmart_testing_two, how="left",
on="d")
# This allows us to work on the data in an aggregate timeseries further on.
Data_walmart_daily
```

Out[ ]:

	d	sales_amount	sell_price	date	snap_CA	snap_TX	snap_WI	event_name_1_
0	d_1	6	4.00	2011-01-29	0	0	0	
1	d_2	15	4.00	2011-01-30	0	0	0	
2	d_3	4	4.00	2011-01-31	0	0	0	
3	d_4	13	4.00	2011-02-01	1	1	0	
4	d_5	8	4.00	2011-02-02	1	0	1	
...	...	...	...	...	...	...	...	
1936	d_1937	12	3.98	2016-05-18	0	0	0	
1937	d_1938	17	3.98	2016-05-19	0	0	0	
1938	d_1939	15	3.98	2016-05-20	0	0	0	
1939	d_1940	9	3.98	2016-05-21	0	0	0	
1940	d_1941	21	3.98	2016-05-22	0	0	0	

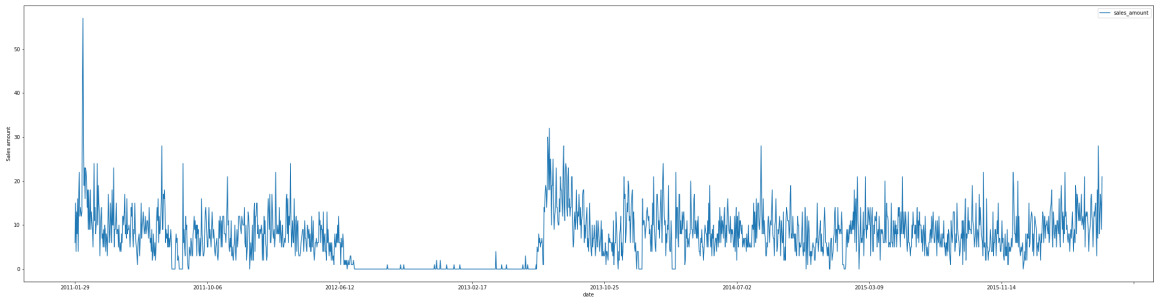
1941 rows × 54 columns

In [ ]:

```
#getting a quick overview of our data. It seems there is a period in the middle where s
ales stop for a while on this product
plt.rcParams["figure.figsize"] = (40,10)
Data_wallmart_daily.plot.line(x="date", y="sales_amount")
plt.ylabel("Sales amount")
```

Out[ ]:

```
Text(0, 0.5, 'Sales amount')
```



It can be observed that our product has a interesting distribution. If Walmart was doing this themselves, they could of course limit the timeframe to when they acctually sell the product

In [ ]:

```
# Setting the date column as index, this will be our unique key, therefore, it might as
well be our index.
Data_walmart_daily.set_index('date', inplace=True)
```

In [ ]:

```
# Dropping the 'd' column, as it is just another index since we dont need it for mergin
g anymore.
Data_walmart_daily = Data_walmart_daily.drop(columns=['d'], axis=1)
```

Supervised Machine Learning

This following part of the notebook will contain a quick runthrough of the data with a linear regressor aswell as a random forest regresseor for supervised machine learning. Due to the sparse variables we expect these models to perform pretty poorly.

In [ ]:

```
#Defining our dataset, for use in quick SML sampling. Y is set to the amount of sales a
cross the state, while X is every other value we prepared.
x_SML = Data_walmart_daily.drop(["sales_amount"], axis=1)
y_SML = Data_walmart_daily.sales_amount
```

In [ ]:

```
#Using the sklearn.train_test_split to split the data in 80, 20 to work in SML. We can
 use SKlearn to do this split here because we are fine with the data getting randomize
d.
x_train, x_test, y_train, y_test = train_test_split(x_SML, y_SML, test_size=0.2, random
_state=21)
```

In [ ]:

```
#Defining our models
RFG_model = RandomForestRegressor()
OLS_model = LinearRegression()
```

In [ ]:

```
#Fitting our models to the split data
OLS_model.fit(x_train, y_train)
RFG_model.fit(x_train, y_train)
```

Out[ ]:

```
RandomForestRegressor()
```

In [ ]:

```
#Finding thr R^2 score of the models
OLS_score = OLS_model.score(x_test, y_test)
RFG_score = RFG_model.score(x_test, y_test)
print("The OLS model's R^2 score is ", OLS_score, "And the RFG models R^2 score is ", R
FG_score)
```

```
The OLS model's R^2 score is  0.3013896826822827 And the RFG models R^2 sc
ore is  0.37806925669314473
```

In [ ]:

```
#Saving the predictions of both models
y_prediction_OLS = OLS_model.predict(x_test)
y_prediction_RFG = RFG_model.predict(x_test)
```

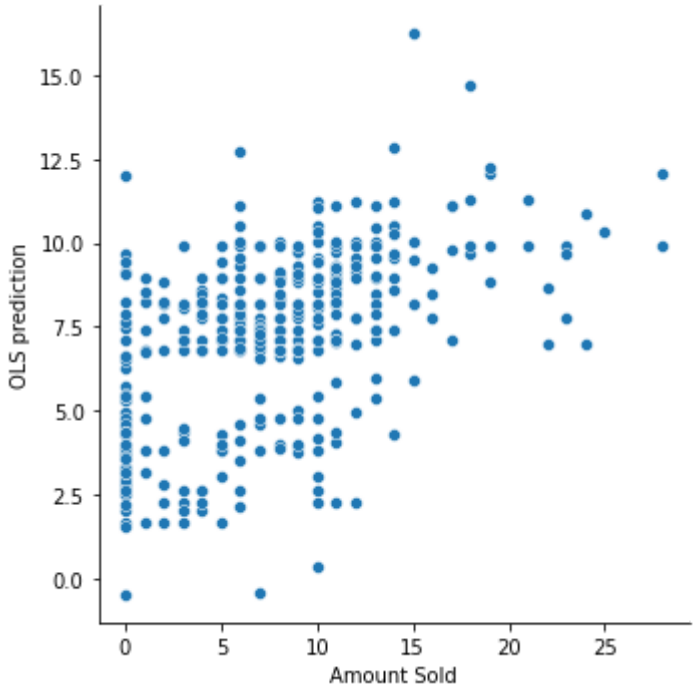
In [ ]:

```
#quick overview of how the linear regression model performed.
sns.relplot(y_test, y_prediction_OLS).set(xlabel="Amount Sold", ylabel= "OLS predictio
n")
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWa
rning:

Pass the following variables as keyword args: x, y. From version 0.12, the
only valid positional argument will be `data`, and passing other arguments
without an explicit keyword will result in an error or misinterpretation.
```

Out[ ]:

```
<seaborn.axisgrid.FacetGrid at 0x7f23e505c350>
```



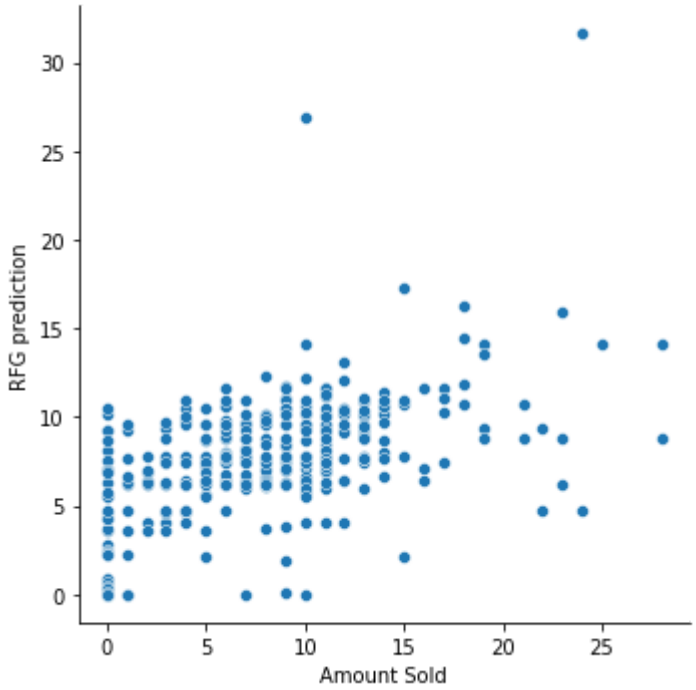
In [ ]:

```
#quick overview of how the random forest regression model performed.
sns.relplot(y_test, y_prediction_RFG).set(xlabel="Amount Sold", ylabel= "RFG predictio
n")
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWa
rning:

Pass the following variables as keyword args: x, y. From version 0.12, the
only valid positional argument will be `data`, and passing other arguments
without an explicit keyword will result in an error or misinterpretation.
```

Out[ ]:

```
<seaborn.axisgrid.FacetGrid at 0x7f23d9dd8a90>
```



In [ ]:

```
#Finding the mean that the model makes by finding the squareroot of the mean sqaure err
or, as we can see it is extremtly high for how most of the data being in the 0-15 range
OLS_error = sqrt(mean_squared_error(y_test,y_prediction_OLS))
RFG_error = sqrt(mean_squared_error(y_test,y_prediction_RFG))
print("The mean error of the OLS model is", OLS_error, "and the mean error for RFG is",
RFG_error)
```

```
The mean error of the OLS model is 4.74688084512352 and the mean error for
RFG is 4.478801814065381
```

So from the above we can observe that the SML models are not very good at handling this issue. This could be because of the sparse data or the complexity of the issue. As we can see the mean error of both is over 4, which is quite large considering the data really only reaches 15-20 at its peak.

Neural Network preprocessing

Now that we have concluded that the SML models are not very suitiable for our problem, we dive into building our neural network

In [ ]:

```
# Defining the test and train size of the dataset.
# We do not use the sklearn train/test split because we are working with a timeseries,
 this should cause the observations to remain in the order of the 'date' index column.
# We do this by choosing 20% length of the dataset for testing
test_size = int(len(Data_walmart_daily) * 0.2)
# As for the training set, the iloc of rows contains the dataset subtracted with the am
ount distrubuted for testing
train = Data_walmart_daily.iloc[:-test_size,:].copy()
# As for the test set, we select the defined 20% for the testset.
test = Data_walmart_daily.iloc[-test_size:,:].copy()
# Showing the shape, we can see the distribution of observations for testing and traini
ng. Keeping the columns for features in the neural network.
print(train.shape, test.shape)
```

```
(1553, 52) (388, 52)
```

In [ ]:

```
# We then split the train and test on x and y, choosing the 0'th column as the predicit
on varible, eg. y_test/y_train the sales amount of the product.
# We use the .values function as to convert the numeric values to a numpy array to use
 later in a scaler function.
x_train = train.iloc[:,0:].values
y_train = train.iloc[:,0].values

x_test = test.iloc[:,0:].values
y_test = test.iloc[:,0].values
```

In [ ]:

```
# Defining a scaler using the sklearn Min Max scaler to scale the data from 0 to 1.
# We can do this on the whole data, the dummy/bool values should retain their values do
ing this, thereby only causing the numeric values of sales price and amount to be scale
d.
# By scaling like this, we can display values as equally distributed numerical values,
 causing the distribution of values to have less varience.
x_scaler = MinMaxScaler(feature_range=(0, 1))
y_scaler = MinMaxScaler(feature_range=(0, 1))
```

In [ ]:

```
# Scaling the training- and testsets with the affordmentioned scaler.
x_train = x_scaler.fit_transform(x_train)
# The reason we reshape the y-train/test is to make it 2 dimensional, which is necessar
y when we are to inverse the scaling.
y_train = y_scaler.fit_transform(y_train.reshape(-1,1))

x_test = x_scaler.transform(x_test)
y_test = y_scaler.transform(y_test.reshape(-1,1))
```

In [ ]:

```
# Choosing the parameters and creating the timeseries generators.
# We are making these timeseries generators as to input in the neural network model, th
ese makes the neural network take timeseries into consideration.
# window_len is the amount of data points that the generator should consider as timeste
ps. We choose 7, as to capture the current and next weeks price update.
# This also determines the amount of observations the model has to look at to predict t
he following value of the next day.
window_len = 7
# Batch size is amount of data points 'batches' we want the model to train on.
batch_size = 2
# n_features should be the amount of variables we want our neural network to test corre
lation on, which is the same as the shape of the 1st column in x_train set.
n_features = x_train.shape[1]
# Defining the time series generator with the affomentioned variables. Making a time s
eries generator for both train and test data.
# Making the train_tsg data as the input of the neural network and the test_tsg as the
 validation data for the neural network.
train_tsg = TimeseriesGenerator(x_train, y_train, length=window_len, batch_size=batch_s
ize)
test_tsg = TimeseriesGenerator(x_test, y_test, length=window_len, batch_size=batch_size
)
```

In [ ]:

```
# Showing the shape of the training timeseries, this shows the 3 dimensional shape with
2 batches, 7 timesteps and 52 features.
train_tsg[0][0].shape
```

Out[ ]:

```
(2, 7, 52)
```

In [ ]:

```
#Building up our model
model = Sequential()
#In the first layer we have chosen 35 as it is slightly more than 2/3rd of the input la
yer.
# This is based on a rule of thumb presented by Jeff Heaton, in "Introduction to Neural
Networks in Java" (2008)
#Alternative source from him: https://web.archive.org/web/20140721050413/http://www.hea
tonresearch.com/node/707
#Following his advice we are also going to keep the neuron amount fairly low to not cau
se the model to overfit.
#We use LSTM layers because we are working with timeseries data. This kind of layer can
input several timestep values at a time to layers below
#This enables the model to compare changes over time.
model.add(LSTM(35, input_shape=(window_len, n_features), return_sequences=True))
model.add(LSTM(20, return_sequences=True))
#We are choosing a dropout layer because it helps with preventing overfitting.
model.add(tf.keras.layers.Dropout(0.3))
model.add(LSTM(5, return_sequences=False))
#Making the dense layer have a linear activiation function, because we want to combine
 the results to a single layer. We would use another activation if we had multiple prod
ucts.
model.add(Dense(1, 'linear'))
#Loss is defined as mean square error, as we can then use the loss function to get a mo
del which is gets the closest to the test data.
#Adam is chosen as, according to (Kingma et al., 2014) it is computaionally efficient a
nd has low memory requirements.
#They also explain it is well suited for problems with a large amount of data / paramet
ers.
#It will become more relevant as we add more products to the model prediction.
model.compile(loss='mean_squared_error', optimizer='adam', metrics=[tf.metrics.MeanAbso
luteError()])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 7, 35)             12320

 lstm_1 (LSTM)               (None, 7, 20)             4480

 dropout (Dropout)           (None, 7, 20)             0

 lstm_2 (LSTM)               (None, 5)                 520

 dense (Dense)               (None, 1)                 6

=================================================================
Total params: 17,326
Trainable params: 17,326
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

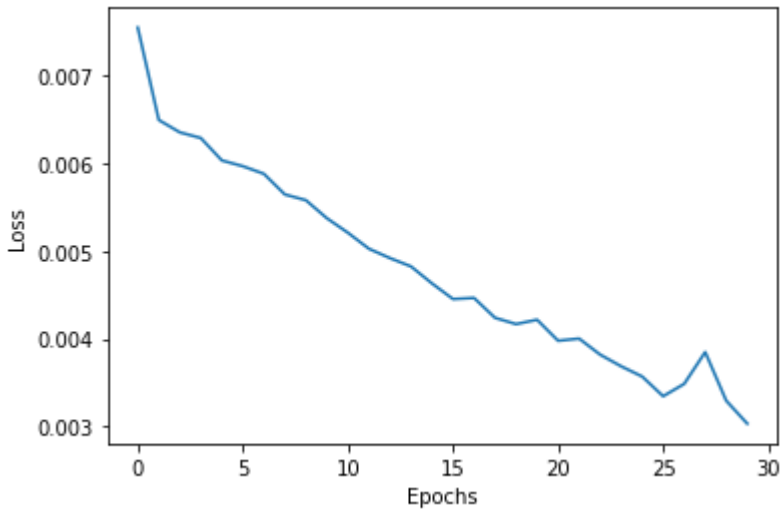
```
# We add an improvement stopper, as to stop the model training if no improvement on MSE is made.
# Patience is the amount of epochs allowed to run without improvement before getting stopped prematurely.
improvement_stop = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5, mode='min')

history = model.fit(train_tsg, epochs=30, validation_data=test_tsg, shuffle=False, callbacks=[improvement_stop], verbose=0)
```

In [ ]:

```
# Displaying the 'loss', or in this case, mean squared error reduction over epochs.
loss_per_epoch = model.history.history['loss']
plt.plot(range(len(loss_per_epoch)),loss_per_epoch);
plt.xlabel("Epochs")
plt.ylabel("Loss")
```

Out[ ]:

```
Text(0, 0.5, 'Loss')
```

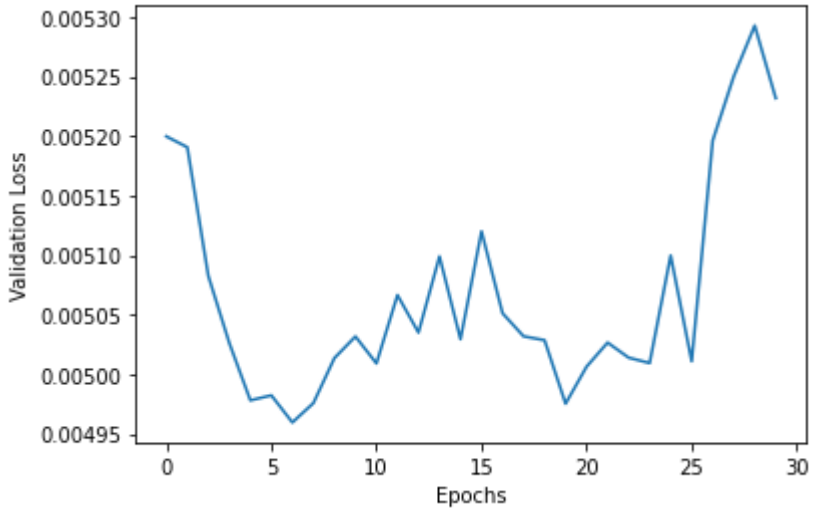


In [ ]:

```
#Displaying the loss again, this time for our validation set
loss_per_epoch = model.history.history['val_loss']
plt.plot(range(len(loss_per_epoch)),loss_per_epoch);
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
```

Out[ ]:

```
Text(0, 0.5, 'Validation Loss')
```



In [ ]:

```
#Displaying a dataframe consisten of our y_test results as compared the model predictio
ns. We do this by inverse transforming the results and our scaled data to compare them.
y_pred_scaled = model.predict(test_tsg)
y_pred = y_scaler.inverse_transform(y_pred_scaled)
y_test_result = y_scaler.inverse_transform(y_test)
#We flatten the result to make them fit into a dataframe.
results = pd.DataFrame({'y_true':y_test_result.flatten()[window_len:],'y_pred':y_pred.f
latten()})
#Here we round the data because all our y_true values are floats anyway, so we are arti
fically setting a 0,50 cutoff for the predictions.
results = results.round(decimals=0)
results
```

Out[ ]:

	y_true	y_pred
0	13.0	10.0
1	8.0	12.0
2	11.0	12.0
3	21.0	12.0
4	8.0	15.0
...	...	...
376	12.0	9.0
377	17.0	11.0
378	15.0	11.0
379	9.0	11.0
380	21.0	9.0

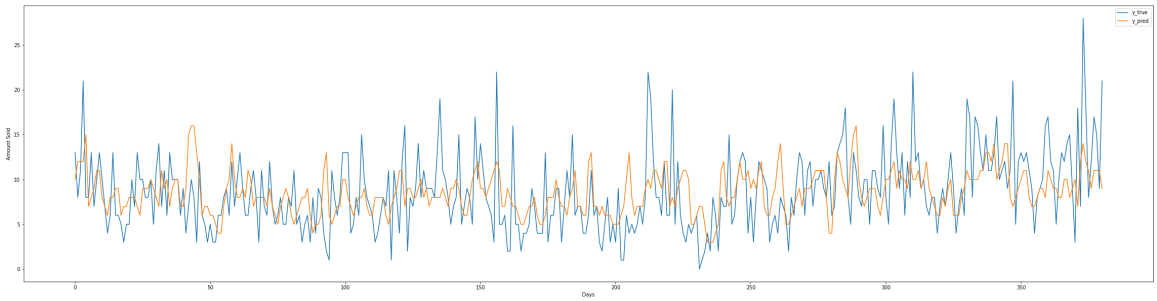
381 rows × 2 columns

In [ ]:

```
#Displaying the 2 graphs up agaianst eachother. It can be observed that the model is de
cent at following the outlines, with the mean being very close to the test set.
plt.rcParams["figure.figsize"] = (40,10)
results.plot()
plt.xlabel("Days")
plt.ylabel("Amount Sold")
```

Out[ ]:

```
Text(0, 0.5, 'Amount Sold')
```



In [ ]:

```
#Here we are displaying the mean error made by the model, as we can see it is considera
bly lower as compared to the earlier SML models we tested on the dataset.
results_error = results
#To find the mean error we have the find the error made in all the columns, we use the
 abs() method to convert them all to positive intigers
results_error["Error"] = abs(results_error["y_true"]-results_error["y_pred"])
#Finding the square root here by lifting to the power of 0,5 here since the sqrt() meth
od doesnt work
results_error["MSE"] = (results_error["Error"]**(1/2)).mean()
#Just to make it look nice
mean_error = results_error["MSE"].mean()
print("The mean error of our initial model, on its test set is", mean_error)
```

```
The mean error of our initial model, on its test set is 1.57577987910122
```

Here we can conclude that the neural network model is considerably more precise at predicting the sales amount, when we look at the mean error. While still not perfect, we do think it could improve with more vast data.

Applying the model with the another product (FOODS_3_800)

In the following part of the notebook we will explore how the model will perform on a different product. As the goal of the problem statement would be to expand this to the whole company. We do realise that the model is currently fitted to a specific product which might not be similar. If we had more capacity we would've been able to feed the model all the data for every product when we were building it.

In [ ]:

```
#This is basically a quick runthrough of everything that happend in our initial "prepro
cessing" part, but this time we are selecting the "FOODS_3_800" and shaping the data to
test the model.
#We hope that products are similear enough that the model is still able to do some work
on it, but first we have to setup the data proper to feed the model we build.
#These are, step for step, the same as previously done when preparing the data.
data_sales_tex = pd.read_csv('/content/data_wallmart/sales_train_evaluation.csv')
data_sell_prices_tx = pd.read_csv('/content/data_wallmart/sell_prices.csv')
discard_items = ["HOUSEHOLD", 'HOBBIES']
data_sell_prices_tx = data_sell_prices_tx[~data_sell_prices_tx.item_id.str.contains('|'
.join(discard_items))]

data_sales_tex = data_sales_tex[data_sales_tex['item_id'].str.contains("FOODS_3_800")]
data_sales_tex = data_sales_tex.melt(['id','item_id','dept_id','cat_id','store_id','sta
te_id'], var_name='d', value_name='sales_amount')#melt to one
data_sales_tex = pd.merge(data_sales_tex, data_calendar, how='inner', on='d')
data_sales_tex = data_sales_tex[data_sales_tex['item_id'].notna()]
data_sales_tex = pd.merge(data_sales_tex, data_sell_prices_tx, how='left', on=['item_i
d', 'wm_yr_wk', 'store_id'])
data_sales_tex = data_sales_tex[data_sales_tex['sell_price'].notna()]
data_sales_tex = pd.get_dummies(data=data_sales_tex, columns=['event_name_1',   'event_
type_1',        'event_name_2', 'event_type_2', 'weekday'])
data_sales_tex = data_sales_tex.drop(columns=['id','item_id','dept_id','cat_id','store_
id','state_id', 'wm_yr_wk', 'wday', 'month','year'], axis=1)
Wallmart_dummy_tx = data_sales_tex


tx_dummy = Wallmart_dummy_tx.groupby("d", as_index=False).agg({"sales_amount": "sum",
                                                               "sell_price":
"mean"})
tx_dummy_two = Wallmart_dummy_tx.drop(labels = ["sales_amount", "sell_price"], axis=1)
tx_dummy_two = tx_dummy_two.groupby("d", as_index=False).agg("mean")
tx_dates = Wallmart_dummy_tx.iloc[:,:3]
tx_dates.drop("sales_amount", axis=1, inplace=True)
tx_dates.drop_duplicates(inplace=True)

tx_weekly = pd.merge(tx_dummy, tx_dates, how="right", on="d")
tx_weekly = pd.merge(tx_weekly, tx_dummy_two, how="left", on="d")
tx_weekly.set_index('date', inplace=True)
tx_weekly = tx_weekly.drop(columns=['d'], axis=1)
```

In [ ]:

```
#Quick overview of the new data after the processing. It looks to be ready as our we ha
ve our 52 columns, one for the y and the rest for the explaining variables.
tx_weekly
```

Out[ ]:

date	sales_amount	sell_price	snap_CA	snap_TX	snap_WI	event_name_1_Chanukah End	eve
2011-01-29	100	1.97	0	0	0	0	
2011-01-30	107	1.97	0	0	0	0	
2011-01-31	104	1.97	0	0	0	0	
2011-02-01	112	1.97	1	1	0	0	
2011-02-02	87	1.97	1	0	1	0	
...	...	...	...	...	...	...	
2016-05-18	32	1.92	0	0	0	0	
2016-05-19	58	1.92	0	0	0	0	
2016-05-20	48	1.92	0	0	0	0	
2016-05-21	61	1.92	0	0	0	0	
2016-05-22	80	1.92	0	0	0	0	

1941 rows × 52 columns

In [ ]:

```
#manually making the x and y data so it stays consistent for our timeseries
X_tx = tx_weekly.iloc[:,0:].values
Y_tx = tx_weekly.iloc[:,0].values
```

In [ ]:

```
#scaling with our same scalers from the initial model.
X_tx = x_scaler.fit_transform(X_tx)
Y_tx = y_scaler.fit_transform(Y_tx.reshape(-1,1))
```

In [ ]:

```
#Making the timeseries
tx_tsg = TimeseriesGenerator(X_tx, Y_tx, length=window_len, batch_size=batch_size)
```

In [ ]:

```
#running it through our model and processing the results as previous.
tx_prediction = model.predict(tx_tsg)
y_tx_pred = y_scaler.inverse_transform(tx_prediction)
y_tx_result = y_scaler.inverse_transform(Y_tx)
results_tx = pd.DataFrame({'y_true':y_tx_result.flatten()[window_len:],'y_pred':y_tx_pr
ed.flatten()})
results_tx = results_tx.round(decimals=0)
results_tx
```

Out[ ]:

	y_true	y_pred
0	135.0	54.0
1	137.0	54.0
2	116.0	53.0
3	73.0	51.0
4	92.0	50.0
...	...	...
1929	32.0	39.0
1930	58.0	43.0
1931	48.0	41.0
1932	61.0	38.0
1933	80.0	33.0

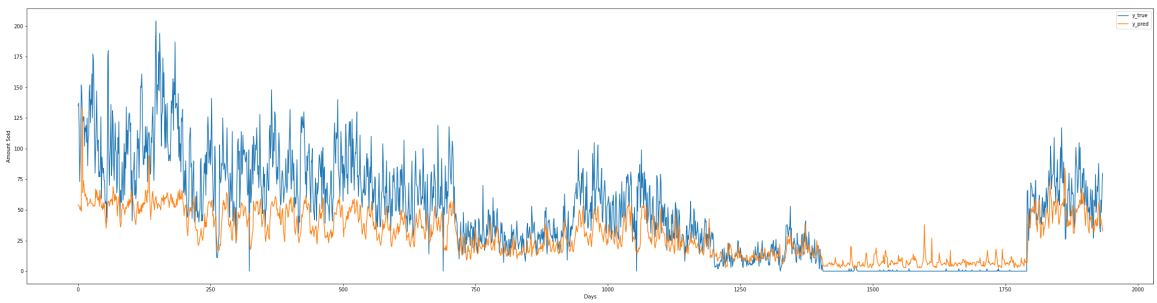
1934 rows × 2 columns

In [ ]:

```
#Comparing the results on a graph. It looks like the model isnt able to do well on this
product as it has a much higher quantity then it got trained with.
#Displaying the 2 graphs up agaianst eachother.
plt.rcParams["figure.figsize"] = (40,10)
results_tx.plot()
plt.xlabel("Days")
plt.ylabel("Amount Sold")
```

Out[ ]:

```
Text(0, 0.5, 'Amount Sold')
```



In [ ]:

```
#Here we are displaying the mean error made by the model, as we can see it is extremly
 high. We suspect this is beacause the model was trained on numbers in the 0-15 range.
results_tx_error = results_tx
results_tx_error["Error"] = abs(results_tx_error["y_true"]-results_tx_error["y_pred"])
results_tx_error["MSE"] = (results_tx_error["Error"]**(1/2)).mean()
mean_error_tx = results_tx_error["MSE"].mean()
print("The mean error of our initial model,tested on another product is", mean_error_tx
)
```

```
The mean error of our initial model,tested on another product is 4.1307955
68353224
```

Conclusion

Rounding up our findings

So through this notebook we have managed to make a model which can decently predict the sales amount of the product the model was built on. That being said though we realise this model cannot really be used outside of this one product, as its very fitted to that one. It is possible to include more products at a time, but since we had a lot of ram issues early on in the data processing part, we chose to be very conservative with the scope of the model, though this perhaps ended up being too conservative.

The final model ended up being pretty good at achieveing it's goal, though we realise the model is very fitted towards that exact product. This is not necessarily an issue, because we could train the model on multiple products if we wish to use it as such, but the comparison at the end, showed that it will not perform well on other products.

Furthermore, we have included some caveats we wish to elaborate upon:

- The exact product is unknown. For instance, it would be interesting group substitutable products and thereby allowing the purchasing agents to buy inventory based on things such as contribution margin.
- Data is "old", it stops at 2016* so new patterns may have emerged. But the model should be easy to transfer to new data.
- Lacking computer power meant we had to reduce the inputs to the models. So we would require more GPU and or RAM.
- Could be interesting to include external data, e.g. inflation, covid-19, tourism, and/or other socio-economic variables.
- The chosen product to build the model does not have many units sold pr day, meaning it will have issues if we want to compare it more popular products, as shown in the final section of the notebook on product Food_3_800.

In [4]:

```
!jupyter nbconvert --to html "/content/MASTER_M3_exam_project_assignment (4).ipynb"
```

```
[NbConvertApp] Converting notebook /content/MASTER_M3_exam_project_assignm
ent (4).ipynb to html
[NbConvertApp] Writing 1884081 bytes to /content/MASTER_M3_exam_project_as
signment (4).html
```