Capstone Project 2 – Product Demand Prediction

Introduction

Accurate demand planning is one of the biggest competitive advantages a company can have in today's fast-paced global economy. Each industry and niche, however, has its own unique demand patterns. How does a company decide which statistical forecasting model to use with which demand pattern?

A 2014 study by Deloitte Consulting surveyed over 400 manufacturing and retail executives throughout the world in regard to high supply chain performance. The survey compared companies on two metrics: 1) inventory turnover and 2) the percentage of on-time and full deliveries. The study found that "supply chain leaders" had a higher percentage of both revenue growth and EBIT (earnings before interest and taxes) margin compared than "supply chain followers". Proper supply chain management can lead not only to improved profitability but also higher market share. It could also be the difference between overall business success and failure.

One of the biggest factors in a high-performing supply chain process is analyzing and planning for your future business needs. A common planning method used by businesses that engage in supply chain management is demand planning. In fact, businesses using big data analytics in demand planning experienced a 425% improvement in order-to-cycle delivery times and more than six times improvement in supply chain efficiency of 10 percent or higher. An Accenture study revealed that businesses that used big data analytics in demand planning experienced a 425% improvement in order-to-cycle delivery times and more than six times improvement in supply chain efficiency of 10 percent or higher.

The Problem

The company has not done any demand planning. Products are ordered based on periodic inventory and sales reports, or when suppliers have specials. Although individual markups are done initially, these are not pro-actively nor continually measured and evaluated. The order "system" is to more or less let store managers place ad-hoc orders for products they feel are running low. If the products are in the company warehouse, the orders get shipped. If not, the order more from the supplier.

When sales are strong, cashflow and profitability seem fine. Lately, company leaders have just noticed a drop in profitability and are wondering if demand planning could be utilized to create a reliable forecast for the business.

This project will analyze current and projected demand for 30 of the company's products in its 76 retail locations as the first step in implementing an overall supply chain management program. Specifically, the main objective of this project will be**: solve the problem of over-stocking and under-stocking of products, by developing a predication model that can predict the demand of products for each store for the next week**. This will be completed as follows:

- Step 1 - Exploratory Data Analysis
- Step 2 – Data Pre-Processing
- Step 3 – Baseline Model & Validation Strategy
- Step 4 – Optimize Prediction Model

Once complete, company leadership will be able to examine company operations and vendor relationships to formalize an official company program.

Step 1 - Exploratory Data Analysis (EDA)

Hypothesis Generation

Before exploring the data, it is important to develop a purpose and a framework for doing so. This will give some context to what we are looking at and why. Since the objective of the project is to develop a predication model, we need to look at the data to see first what it looks like, then what are the characteristics, features and relationships between the data that will help us be able to predict product demand at each store. In general, what we are looking for is **factors that might affect the target variable in our model, which in this case is product demand at each store**.

We are going to develop some hypotheses in regards to our objective before we look at the available data, because we want to consider all factors that could potentially impact product demand and we do not want to be biased by what data is already available.

Data Summary

The data used in this project was acquired from the company. They provided three datasets:

1) sales.csv – contains 232,287 sales records with the UPC code of the product sold, the sales date, the store ID where the product was sold, the sales price, the base price, whether the product was on promotion for the week of the sale (1 or 0), whether the product was in the in-store circular (1 or 0) and the number of units sold for each week;
2) product_data.csv – contains the product description, manufacturer, product category and sub-category, product size and UPC for 30 products; and
3) store_data.csv – contains the store ID, store name, city, state, MSA code, market segment type of store, number of store parking spaces, store sales area square footage and average weekly baskets, for each of the 76 store locations.

We will first explore each dataset separately.

Sales Dataframe EDA

The following is a summary of the columns of the sales.csv dataset, which as mentioned earlier contains 232,287 rows of data.

Sales Data: ('sales.csv')

- **WEEK_END_DATE** - week ending date of sales report
- **STORE_NUM** - store number where sale was made
- **UPC** - (Universal Product Code) product specific identifier
- **BASE_PRICE** - base price of item
- **DISPLAY** - whether product was a part of in-store promotional display (1-Yes, 0-No)
- **FEATURE** - whether product was in in-store circular (1-Yes, 0-No)
- **UNITS** - units sold (target)

The first step in understanding the data is to take a quick look at the structure and data types.

Sales Data ('sales.csv')

```
# Print first 5 rows of the sales dataframe
sales.head()
```

	WEEK_END_DATE	STORE_NUM	UPC	BASE_PRICE	FEATURE	DISPLAY	UNITS
0	14-Jan-09	367	1111009477	1.57	0	0	13
1	14-Jan-09	367	1111009497	1.39	0	0	20
2	14-Jan-09	367	1111085319	1.88	0	0	14
3	14-Jan-09	367	1111085345	1.88	0	0	29
4	14-Jan-09	367	1111085350	1.98	0	0	35

```
# Check datatypes of columns in sales dataframe
sales.dtypes
```

```
WEEK_END_DATE     object
STORE_NUM          int64
UPC                int64
BASE_PRICE       float64
FEATURE            int64
DISPLAY            int64
UNITS              int64
dtype: object
```

Two issues that stand out at first are:

- WEEK_END_DATE has been imported as an object, but it is a datetime variable. This needs to be converted.
- The store number and product codes have been imported as integers, but these are categorical variables. This needs to be fixed as well.

Other issues/questions are:

- Datetime variable
  - What are the start and end dates?
  - Are these periodic intervals and are they regular?
  - Are there any missing data points?
- Numerical Variables ('BASE_PRICE' and 'UNITS')
  - Need to check the distribution of numerical variables.
  - Also need to check if there are any outliers or missing values.
- Categorical Variables ('FEATURE' and 'DISPLAY')
  - Check the unique values for categorical variables
  - Are there any missing values?
  - Are there any variables with high cardinality / sparsity?

WEEK_END_DATE

- This variable was converted to a date time object.
- The sales data is for 142 weeks, based on the number of unique WEEK_END_DATE's in the sales file, starting on January 13, 2016 and ending September 26, 2018.
- No dates are missing from this period.
- All of the dates fall on Wednesday, which appears to be the date that the sales reports are generated.

STORE_NUM and UPC

- There are no missing values in either variable.
- All 76 stores have reported sales transactions, although not the same number.
- The number of transactions reported by each store range from a low of 1,676 up to a high of 4,098.
- All stores reported selling at least one product each week (142 weeks x 76 stores = 10,792 records, which is the number of unique records in the data.
- There were 30 unique UPC codes found in the data which means that each product was sold, with the minimum number sold of 975 and a maximum of 10,790.
- With 30 products, 76 stores and 142 weeks, if every product was sold at every store at least once every week, there would be 323,760 rows of data. Since we only have 232,287 rows, not every product was sold at every store every week. This did occur 71.7% of the time.
- The average number of unique products sold each week is 22.

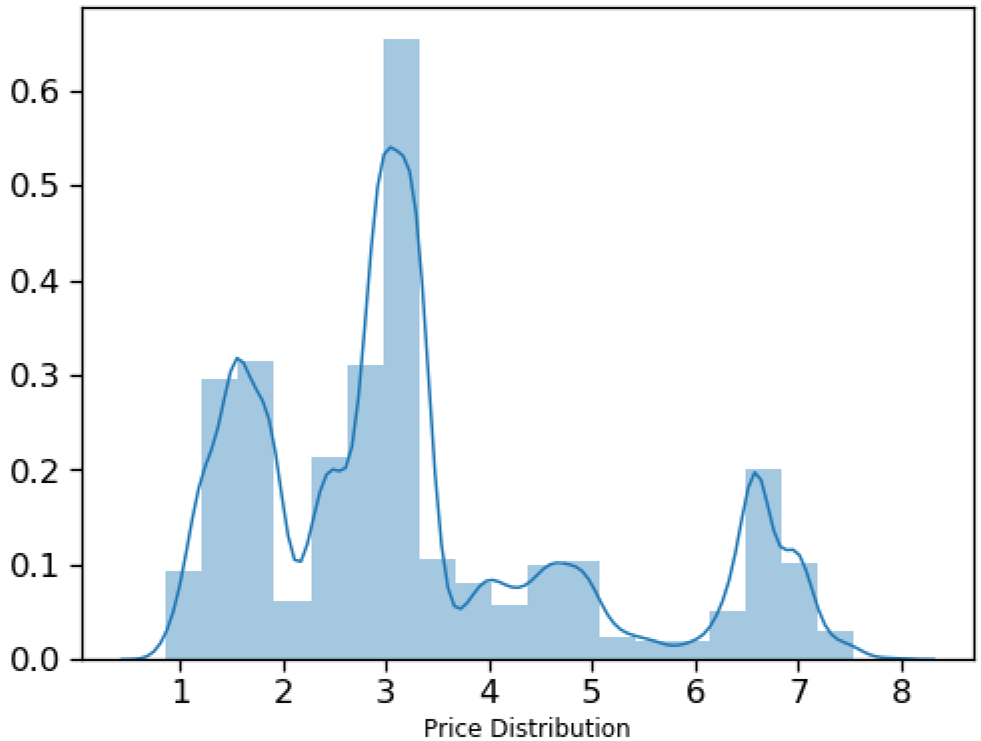
BASE_PRICE

- There are no missing values.
- The basic statistics and distribution of the variable are as follows:

```
[29]: # Examine basic statistical details of BASE_PRICE variable.
      sales['BASE_PRICE'].describe()
```

```
[29]: count    232275.000000
      mean          3.345204
      std           1.678181
      min           0.860000
      25%           1.950000
      50%           2.990000
      75%           4.080000
      max           7.890000
      Name: BASE_PRICE, dtype: float64
```

[30]:
```
# distribution of Base Price variable
plt.figure(figsize=(8,6))
sns.distplot((sales['BASE_PRICE'].values), bins=20, kde=True)
plt.xlabel('Price Distribution', fontsize=12)
plt.show()
```



- There are no extreme values in the BASE_PRICE variable.
- The range for base price is 0.86 to 7.89, with an average of 3.35.

FEATURE and DISPLAY

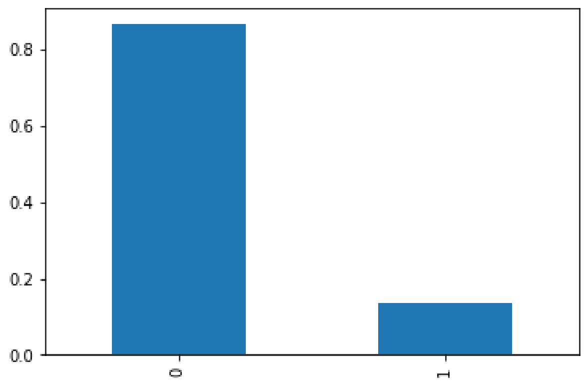
- There are no missing values in either variable.
- Both variables were imported as integer values, with both having either a ‘1’ for ‘Yes’ or ‘0’ for ‘No’.
- The value counts for each variable are shown here:

```python
# Examine values for 'FEATURE'.
sales['DISPLAY'].value_counts(normalize=True)
```

```
0    0.864998
1    0.135002
Name: DISPLAY, dtype: float64
```

```python
sales['DISPLAY'].value_counts(normalize=True).plot('bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2e2b402f688>
```



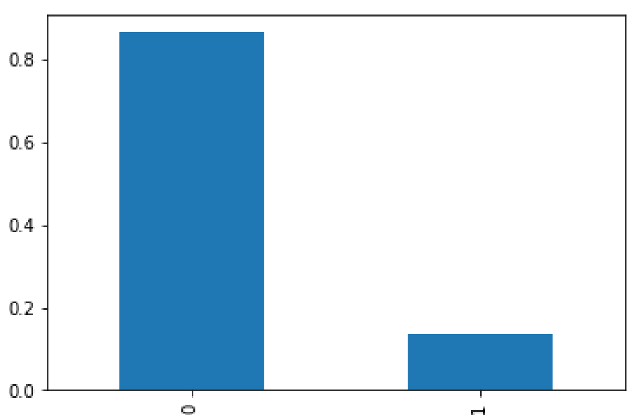
- Approximately 13.5 percent of products are on display

```python
# Examine values for 'DISPLAY'.
sales['DISPLAY'].value_counts(normalize=True)
```

```
0    0.864998
1    0.135002
Name: DISPLAY, dtype: float64
```

```python
sales['DISPLAY'].value_counts(normalize=True).plot('bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2e2b3e4e9c8>
```



- Approximately 13.5 percent of products are on display

- The cross-tab table for FEATURE and DISPLAY is:

```
pd.crosstab(sales['FEATURE'], sales['DISPLAY']).apply(lambda r: r/len(sales), axis=1)
```

DISPLAY	0	1
FEATURE		
0	0.821824	0.078287
1	0.043175	0.056714

UNITS

- The basic statistical details for the UNITS variable are:

```
# Examine basic statistical details of UNITS variable.
sales['UNITS'].describe()
```

```
count    232287.000000
mean         28.063525
std          35.954341
min           0.000000
25%           9.000000
50%          18.000000
75%          34.000000
max        1800.000000
Name: UNITS, dtype: float64
```

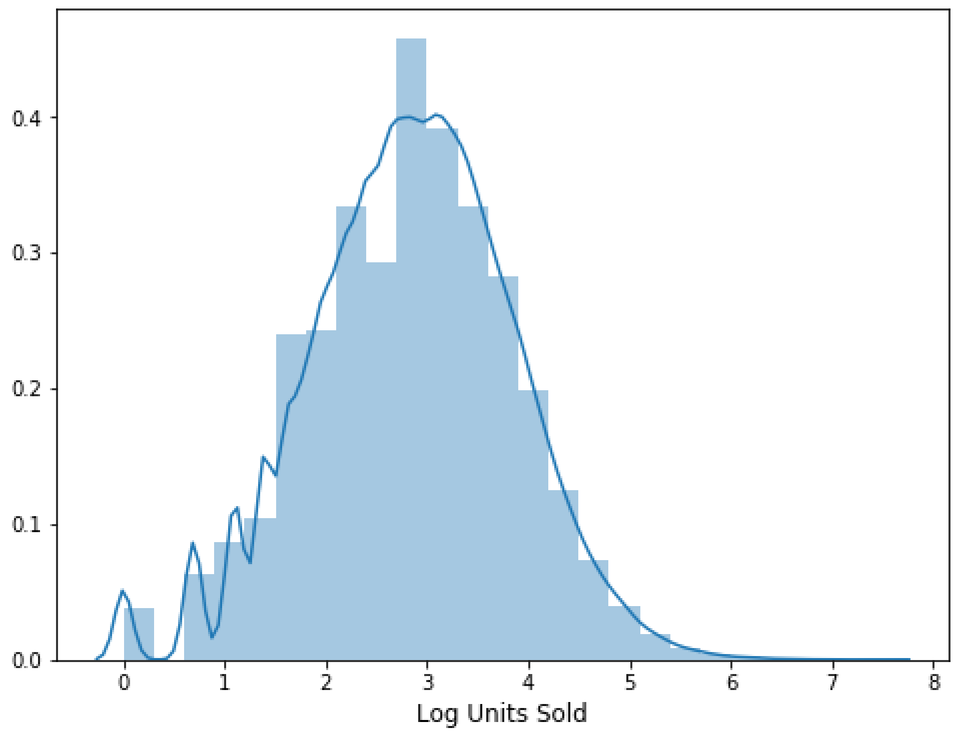
- The range of values is fairly high
- The minimum number of units sold is 0, with a maximum of 1,800.
- There is a huge difference between the 75th percentile and the max value, which indicates the presence of outliers.

- There was only one row with 0 units sold; this row was dropped
- There were four rows of data with more than 1,000 units sold. To reduce the effect of outliers and for better visualization of the distribution, we plotted the log transformation for UNITS:

```
sales[sales['UNITS'] > 1000]
```

	WEEK_END_DATE	STORE_NUM	UPC	PRICE	BASE_PRICE	FEATURE	DISPLAY	UNITS
7893	2016-02-10	24991	1600027527	1.67	3.19	1	0	1006
7960	2016-02-10	25027	1600027527	1.64	3.19	1	1	1800
9597	2016-02-17	25027	1600027527	1.60	3.19	0	1	1054
11209	2016-02-24	25027	1600027527	1.64	3.19	1	1	1136

```python
# log transformed UNITS column
plt.figure(figsize=(8,6))
sns.distplot(np.log(sales['UNITS'].values), bins=25, kde=True)
plt.xlabel('Log Units Sold', fontsize=12)
plt.show()
```



- After log transformation, the distribution looks closer to a normal distribution

Products Dataframe EDA

The following is a summary of the columns of the product_data.csv dataset, which as mentioned earlier contains 30 rows of data.

Product Data: ('product_data.csv')

- **UPC** - (Universal Product Code) product specific identifier
- **DESCRIPTION** - product description
- **MANUFACTURER** - product manufacturer/supplier
- **CATEGORY** - product category
- **SUB_CATEGORY** - product sub-category
- **PRODUCT_SIZE** - package size/quantity

```python
# Print first five rows of product data
products.head()
```

	UPC	DESCRIPTION	MANUFACTURER	CATEGORY	SUB_CATEGORY	PRODUCT_SIZE
0	1111009477	PL MINI TWIST PRETZELS	PRIVATE LABEL	BAG SNACKS	PRETZELS	15 OZ
1	1111009497	PL PRETZEL STICKS	PRIVATE LABEL	BAG SNACKS	PRETZELS	15 OZ
2	1111009507	PL TWIST PRETZELS	PRIVATE LABEL	BAG SNACKS	PRETZELS	15 OZ
3	1111038078	PL BL MINT ANTSPTC RINSE	PRIVATE LABEL	ORAL HYGIENE PRODUCTS	MOUTHWASHES (ANTISEPTIC)	500 ML
4	1111038080	PL ANTSPTC SPG MNT MTHWS	PRIVATE LABEL	ORAL HYGIENE PRODUCTS	MOUTHWASHES (ANTISEPTIC)	500 ML

```python
products.dtypes
```

```
UPC              int64
DESCRIPTION     object
MANUFACTURER    object
CATEGORY        object
SUB_CATEGORY    object
PRODUCT_SIZE    object
dtype: object
```

Categorical Variables

All of the variables in the Products dataframe are categorical, except for UPC code which is just the identifier which will be used to join with the Sales dataframe. As such, the following issues need to be addressed:

- Check unique values.
- Are there any missing values?
- Are there any variables with high cardinality or sparsity?

UPC

- In examining the 'UPC' variable, we found 30 unique values, which we validated were identical to the 'UPC' variable values in the Sales dataframe.

CATEGORY

- There are no missing values.
- The 'CATEGORY' variable has four unique values. The details for which are shown below:

CATEGORY

```
# Number and list of unique categories in the product data
products['CATEGORY'].nunique(), products['CATEGORY'].unique()
```

```
(4, array(['BAG SNACKS', 'ORAL HYGIENE PRODUCTS', 'COLD CEREAL',
        'FROZEN PIZZA'], dtype=object))
```

```
products['CATEGORY'].isnull().sum()
```

```
0
```

```
products['CATEGORY'].value_counts()
```

```
COLD CEREAL              9
BAG SNACKS               8
FROZEN PIZZA             7
ORAL HYGIENE PRODUCTS    6
Name: CATEGORY, dtype: int64
```

- There are four product categories:
  - BAG SNACKS
  - ORAL HYGIENE PRODUCTS
  - COLD CEREAL
  - FROZEN PIZZA

- There are 9 products with the category 'Cold Cereal', 8 products labeled 'Bag snacks', 7 with category 'Frozen Pizza' and 6 'Oral Hygiene' Products.

SUB_CATEGORY

- There are no missing values.
- The 'SUB_CATEGORY' variable has four unique values. The details for which are shown below:

SUB_CATEGORY

```
# Check for null values.
products['SUB_CATEGORY'].isnull().sum()
```

```
0
```

```
products['SUB_CATEGORY'].nunique()
```

```
7
```

```
# Display subcategories for each category
products[['CATEGORY','SUB_CATEGORY']].drop_duplicates().sort_values(by = 'CATEGORY')
```

	CATEGORY	SUB_CATEGORY
0	BAG SNACKS	PRETZELS
5	COLD CEREAL	ALL FAMILY CEREAL
6	COLD CEREAL	ADULT CEREAL
19	COLD CEREAL	KIDS CEREAL
8	FROZEN PIZZA	PIZZA/PREMIUM
3	ORAL HYGIENE PRODUCTS	MOUTHWASHES (ANTISEPTIC)
16	ORAL HYGIENE PRODUCTS	MOUTHWASH/RINSES AND SPRAYS

The sub-categories give additional detail about the products.

- Cereal has 3 sub categories, differentiating on the age group.
- Oral hygiene products have 2 sub categories, antiseptic and rinse/spray.
- Bag Snacks & Frozen Pizza have just 1 sub category.

PRODUCT_SIZE

- The following is a summary of the PRODUCT_SIZE variable:

```python
# Examine unique category, sub-category and product size combinations.
products[['CATEGORY','SUB_CATEGORY','PRODUCT_SIZE']].drop_duplicates().sort_values(by = 'CATEGORY')
```

	CATEGORY	SUB_CATEGORY	PRODUCT_SIZE
0	BAG SNACKS	PRETZELS	15 OZ
14	BAG SNACKS	PRETZELS	16 OZ
25	BAG SNACKS	PRETZELS	10 OZ
6	COLD CEREAL	ADULT CEREAL	20 OZ
7	COLD CEREAL	ALL FAMILY CEREAL	18 OZ
19	COLD CEREAL	KIDS CEREAL	15 OZ
20	COLD CEREAL	KIDS CEREAL	12.2 OZ
5	COLD CEREAL	ALL FAMILY CEREAL	12.25 OZ
13	COLD CEREAL	ALL FAMILY CEREAL	12 OZ
8	FROZEN PIZZA	PIZZA/PREMIUM	32.7 OZ
9	FROZEN PIZZA	PIZZA/PREMIUM	30.5 OZ
10	FROZEN PIZZA	PIZZA/PREMIUM	29.6 OZ
24	FROZEN PIZZA	PIZZA/PREMIUM	22.7 OZ
21	FROZEN PIZZA	PIZZA/PREMIUM	29.8 OZ
23	FROZEN PIZZA	PIZZA/PREMIUM	28.3 OZ
3	ORAL HYGIENE PRODUCTS	MOUTHWASHES (ANTISEPTIC)	500 ML
16	ORAL HYGIENE PRODUCTS	MOUTHWASH/RINSES AND SPRAYS	1 LT
17	ORAL HYGIENE PRODUCTS	MOUTHWASHES (ANTISEPTIC)	1 LT

- In reviewing the SUB_CATEGORY with PRODUCT_SIZE, there are no combinations that indicate that SUB_CATEGORY is an indicator of size.

To summarize:

- Bag Snacks has 1 sub-category and 3 product sizes.
- Oral Hygiene has 2 sub-categories and 2 size options.
- Frozen Pizza has only 1 sub-category and 6 different package sizes.
- Cold Cereal has 3 sub-categories, and 6 size options.

DESCRIPTION

- There are no missing values.
- There are 29 unique values, with one description (GM CHEERIOS) being used twice.
- GM CHEERIOS uses the same description for two product sizes (18 OZ & 12 OZ).

MANUFACTURER

- There are no missing values.
- There are 9 unique values, broken down as follows:

```python
products['MANUFACTURER'].nunique()
```

9

```python
# displaying the list of manufacturers against the 4 categories
temp = products[['CATEGORY','MANUFACTURER']].drop_duplicates()
pd.crosstab([temp['CATEGORY']], temp['MANUFACTURER'])
```

MANUFACTURER CATEGORY	FRITO LAY	GENERAL MI	KELLOGG	P & G	PRIVATE LABEL	SNYDER S	TOMBSTONE	TONYS	WARNER
BAG SNACKS	1	0	0	0	1	1	0	0	0
COLD CEREAL	0	1	1	0	1	0	0	0	0
FROZEN PIZZA	0	0	0	0	1	0	1	1	0
ORAL HYGIENE PRODUCTS	0	0	0	1	1	0	0	0	1

- With 4 unique categories of Products, each category has three different manufacturers.
- Each category has a manufacturer identified as 'private label' along with 2 other manufacturers.

Stores Dataframe EDA

The following is a summary of the columns of the store_data.csv dataset, which as mentioned earlier contains 76 rows of data.

Store Data: ('store_data.csv')

- **STORE_ID** - store number
- **STORE_NAME** - Name of store
- **ADDRESS_CITY_NAME** - city
- **ADDRESS_STATE_PROV_CODE** - state
- **MSA_CODE** - (Metropolitan Statistical Area) Based on geographic region and population density
- **SEG_VALUE_NAME** - Store Segment Name
- **PARKING_SPACE_QTY** - number of parking spaces in the store parking lot
- **SALES_AREA_SIZE_NUM** - square footage of store
- **AVG_WEEKLY_BASKETS** - average weekly baskets sold in the store

We will examine Numerical and Categorical variables separately.

Numerical Variables

- Are there any missing values in the variables?
- What does the distribution look like?
- Are there any extreme/outlier values?

Categorical Variables

- Check the unique values for categorical variables.
- Are there any missing values in the variables?
- Are there any variables with high cardinality or sparsity?

STORE_ID & STORE_NAME

STORE_ID is a key variable and will be used to join with the Sales dataframe later.

STORE_NAME is a categorical value that represents the city that the store is located in.

- There are 76 unique values in STORE_ID
- There are 72 unique values in STORE_NAME
- Some store names are being repeated, which means there are some cities with more than one store.

```
# number of store names repeating
stores['STORE_NAME'].value_counts()
```

```
HOUSTON              4
MIDDLETOWN           2
SILVERLAKE           1
MAGNOLIA             1
ANTOINE TOWN CENTER  1
                    ..
FLOWER MOUND         1
CYPRESS              1
MIAMI TOWNSHIP       1
BLUE ASH             1
KROGER JUNCTION S/C  1
Name: STORE_NAME, Length: 72, dtype: int64
```

```
stores.loc[stores['STORE_NAME'] == 'HOUSTON']
```

	STORE_ID	STORE_NAME	ADDRESS_CITY_NAME	ADDRESS_STATE_PROV_CODE	MSA_CODE	SEG_VALUE_NAME	PARKING_SPACE_QTY	SALES_AREA_SIZE_NUM	AVG_WEEKLY_BASKETS
3	623	HOUSTON	HOUSTON	TX	26420	MAINSTREAM	NaN	46930	36741
9	2513	HOUSTON	HOUSTON	TX	26420	UPSCALE	NaN	61833	32423
54	21485	HOUSTON	KATY	TX	26420	MAINSTREAM	NaN	46369	26472
59	23327	HOUSTON	HOUSTON	TX	26420	MAINSTREAM	NaN	50722	30258

```
stores.loc[stores['STORE_NAME'] == 'MIDDLETOWN']
```

	STORE_ID	STORE_NAME	ADDRESS_CITY_NAME	ADDRESS_STATE_PROV_CODE	MSA_CODE	SEG_VALUE_NAME	PARKING_SPACE_QTY	SALES_AREA_SIZE_NUM	AVG_WEEKLY_BASKETS
50	21221	MIDDLETOWN	MIDDLETOWN	OH	17140	VALUE	NaN	48128	17010
74	28909	MIDDLETOWN	MIDDLETOWN	OH	17140	MAINSTREAM	NaN	85876	28986

- We see that four stores are named ‘Houston’ and two are named ‘Middletown’. Each store has a different segment value, location and/or sales area size, so they are in fact different stores.
- There are no missing values.

ADDRESS_CITY_NAME and ADDRESS_STATE_PROV_CODE

- There are no missing values.
- There are 51 unique city names in 4 states.

```
# Check for null values.
stores[['ADDRESS_STATE_PROV_CODE', 'ADDRESS_CITY_NAME']].isnull().sum()
```

```
ADDRESS_STATE_PROV_CODE    0
ADDRESS_CITY_NAME          0
dtype: int64
```

```
# How many unique cities in which states?
stores[['ADDRESS_STATE_PROV_CODE', 'ADDRESS_CITY_NAME']].nunique()
```

```
ADDRESS_STATE_PROV_CODE     4
ADDRESS_CITY_NAME          51
dtype: int64
```

- The number of stores per state with some interesting findings:

```
stores.groupby(['ADDRESS_STATE_PROV_CODE'])['STORE_ID'].count()
```

```
ADDRESS_STATE_PROV_CODE
IN     1
KY     4
OH    30
TX    41
Name: STORE_ID, dtype: int64
```

- Each store has a unique store ID
- Most stores are from Ohio and Texas ~93%
- Few from Kentucky and Indiana ~7%

```
stores['ADDRESS_CITY_NAME'].value_counts()
```

```
CINCINNATI          9
HOUSTON             8
MIDDLETOWN          3
COVINGTON           2
MAINEVILLE          2
LOVELAND            2
HAMILTON            2
MCKINNEY            2
DAYTON              2
KATY                2
SUGAR LAND          2
KETTERING           1
DALLAS              1
SPRINGFIELD         1
MAGNOLIA            1
ARLINGTON           1
THE WOODLANDS       1
DICKINSON           1
CLUTE               1
```

- Cincinnati and Houston have the most stores (partial list, sorted from most to least).
- 11 cities have more than one store.

MSA_CODE

- There are no missing values.
- There are 9 unique MSA code values
- The top 3 MSA codes are ‘17140’ with 29, ‘26420’ with 21, and ‘19100’ with 17.

```
stores['MSA_CODE'].nunique(), stores['MSA_CODE'].unique()
```

```
(9, array([17140, 19100, 26420, 17780, 47540, 43300, 19380, 13140, 44220],
       dtype=int64))
```

```
stores['MSA_CODE'].value_counts()
```

```
17140    29
26420    21
19100    17
19380     4
13140     1
47540     1
44220     1
43300     1
17780     1
Name: MSA_CODE, dtype: int64
```

```
(stores.groupby(['MSA_CODE', 'ADDRESS_STATE_PROV_CODE'])['STORE_ID'].count())
```

```
MSA_CODE  ADDRESS_STATE_PROV_CODE
13140     TX                          1
17140     IN                          1
          KY                          4
          OH                         24
17780     TX                          1
19100     TX                         17
19380     OH                          4
26420     TX                         21
43300     TX                          1
44220     OH                          1
47540     OH                          1
Name: STORE_ID, dtype: int64
```

- These codes are assigned based on the geographical location and population density.
- 17140 is present in all three except Texas (which has a different geographical region)

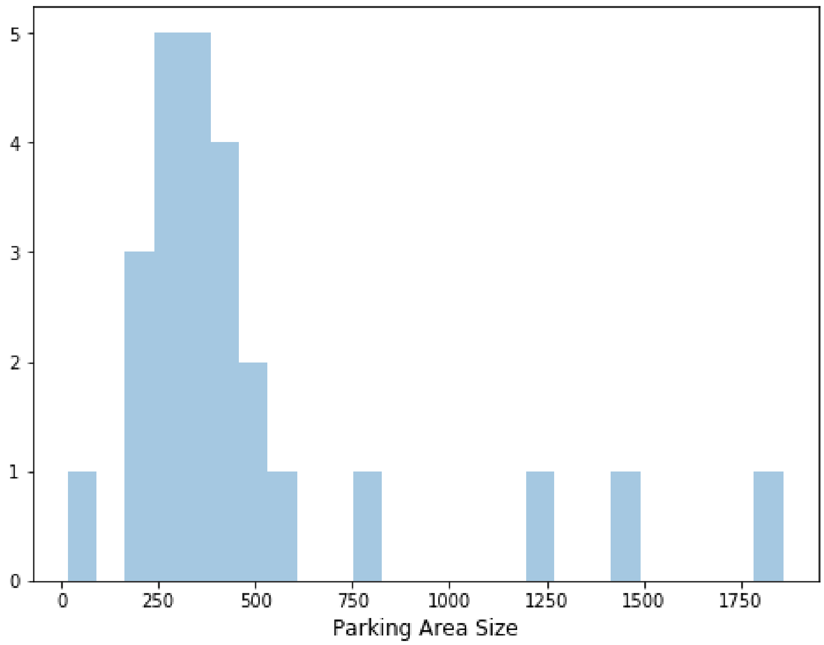
PARKING_SPACE_QTY and SALES_AREA_SIZE_NUM

- Of the 76 stores, parking area is missing for 51 of them.

```
stores[['PARKING_SPACE_QTY', 'SALES_AREA_SIZE_NUM']].isnull().sum()
```

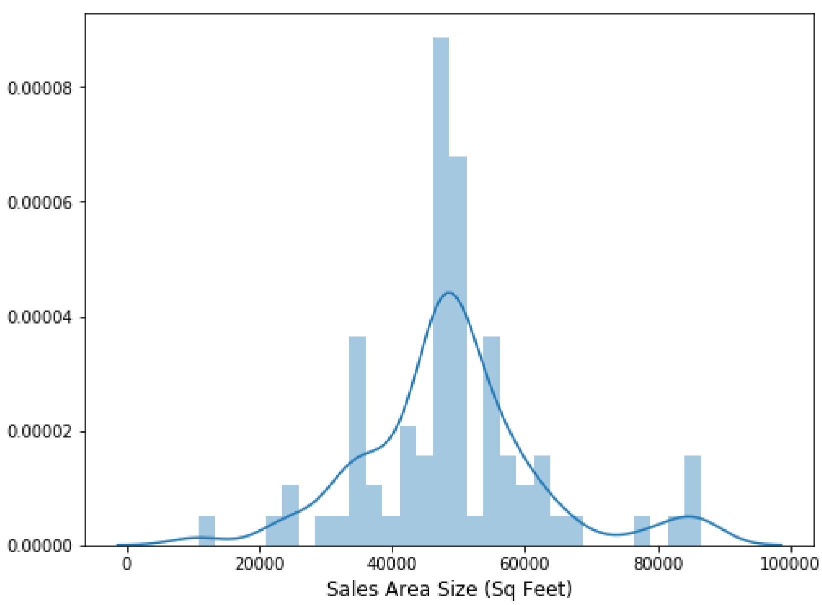
```
PARKING_SPACE_QTY      51
SALES_AREA_SIZE_NUM     0
dtype: int64
```

```python
plt.figure(figsize=(8,6))
sns.distplot(stores['PARKING_SPACE_QTY'], bins=25, kde=False)
plt.xlabel('Parking Area Size', fontsize=12)
plt.show()
```



- About 15 stores have between 250-500 parking spaces.

```python
plt.figure(figsize=(8,6))
sns.distplot(stores['SALES_AREA_SIZE_NUM'], bins=30, kde=True)
plt.xlabel('Sales Area Size (Sq Feet)', fontsize=12)
plt.show()
```



- Most stores have between 30,000 and 70,000 square feet of sales area.
- Only a few of the stores have less than 30,000 square feet or more than 90,000 square feet of sales area.

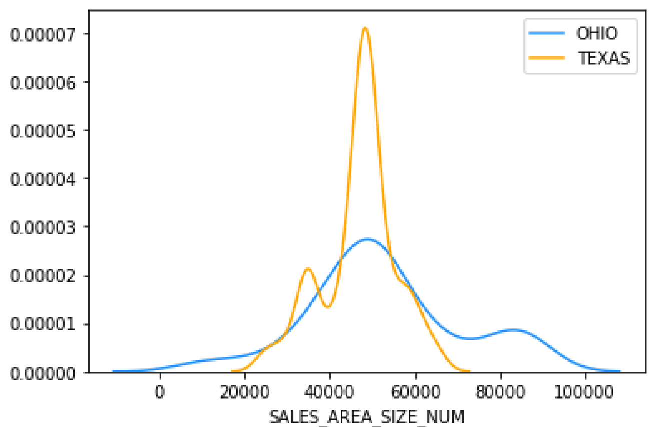
```
(stores.groupby(['ADDRESS_STATE_PROV_CODE'])['SALES_AREA_SIZE_NUM'].mean()).sort_values(ascending=False)
```

```
ADDRESS_STATE_PROV_CODE
IN    58563.000000
OH    52691.200000
TX    46920.902439
KY    39855.500000
Name: SALES_AREA_SIZE_NUM, dtype: float64
```

```
state_oh = stores.loc[stores['ADDRESS_STATE_PROV_CODE'] == 'OH']
state_tx = stores.loc[stores['ADDRESS_STATE_PROV_CODE'] == 'TX']

sns.distplot(state_oh['SALES_AREA_SIZE_NUM'], hist=False,color= 'dodgerblue', label= 'OHIO')
sns.distplot(state_tx['SALES_AREA_SIZE_NUM'], hist=False,  color= 'orange', label= 'TEXAS')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x173d0ca48c8>
```



- Indiana has the largest mean store sales area size; Kentucky has the smallest.
- Texas has some of the largest stores, but also some smaller ones too, which brings the Texas overall store sales area size down compared to Indiana and Ohio.
- Ohio's stores are more evenly distributed.

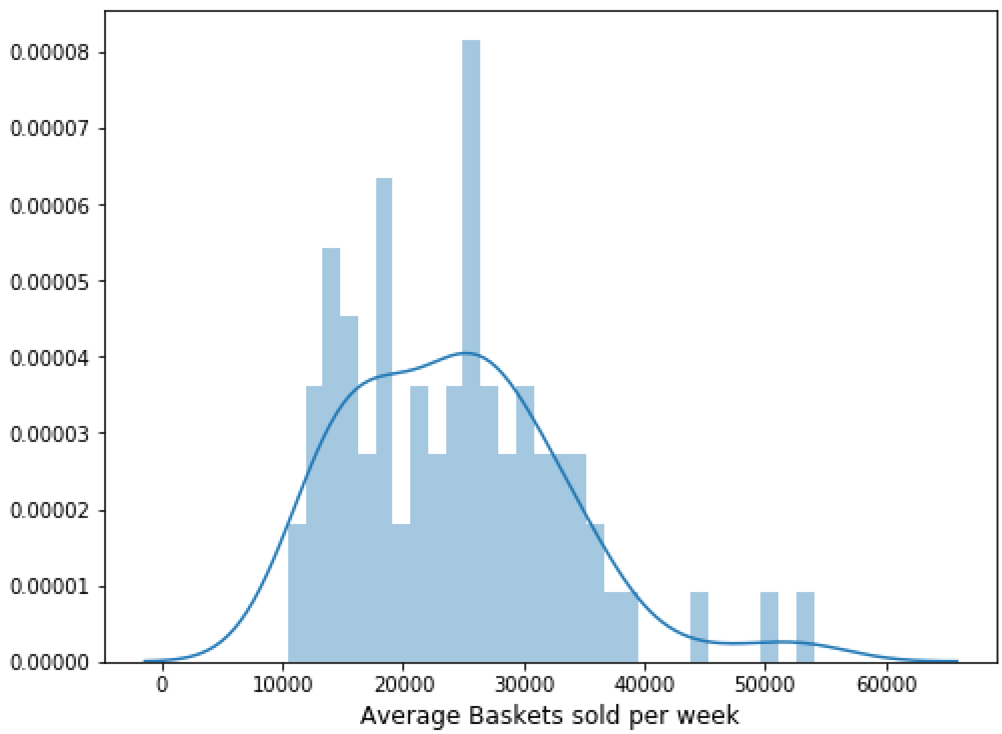
AVG_WEEKLY_BASKETS

- There are no missing values.
- The basic statistics for Average Baskets sold per week and associated distribution are as follows:

```
stores['AVG_WEEKLY_BASKETS'].describe()
```

```
count       76.000000
mean     24226.921053
std       8863.939362
min      10435.000000
25%      16983.500000
50%      24667.500000
75%      29398.500000
max      54053.000000
Name: AVG_WEEKLY_BASKETS, dtype: float64
```

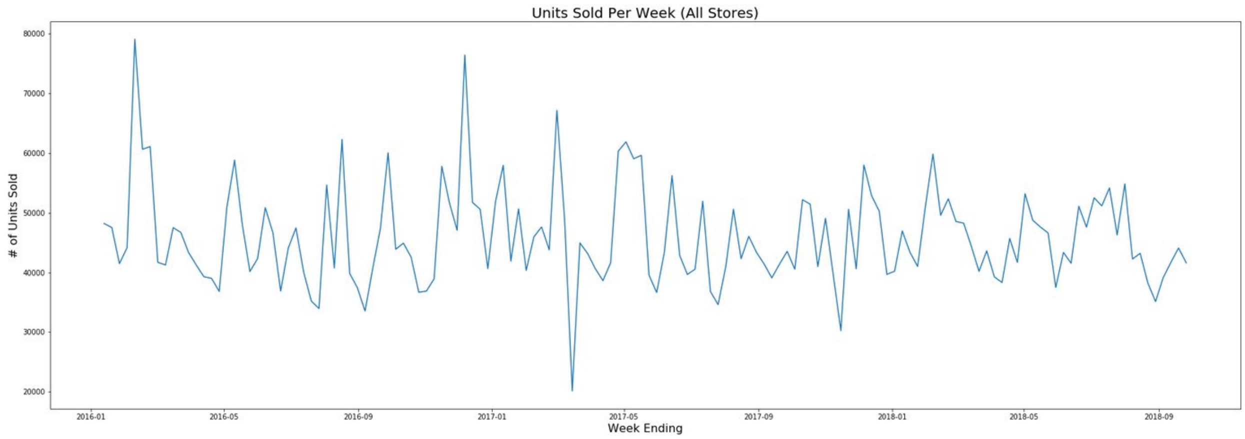
```
plt.figure(figsize=(8,6))
sns.distplot(stores['AVG_WEEKLY_BASKETS'], bins=30, kde=True)
plt.xlabel('Average Baskets sold per week', fontsize=12)
plt.show()
```



Trends and/or Seasonal Patterns in Product Sales

In order to look for overall sales trends and seasonal patterns, we merged the store and product datasets. We are looking to see if there are any trends or patterns in total units sold per week.

Units Sold Per Week at Company Level

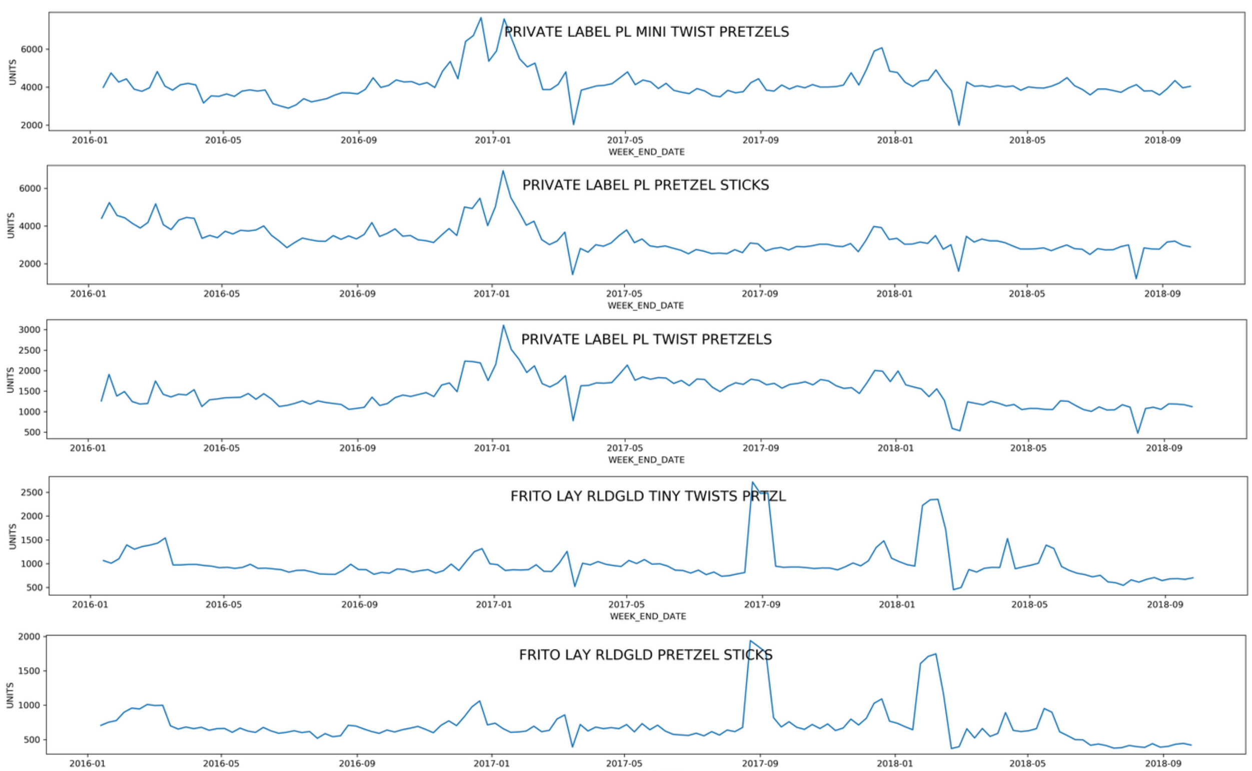


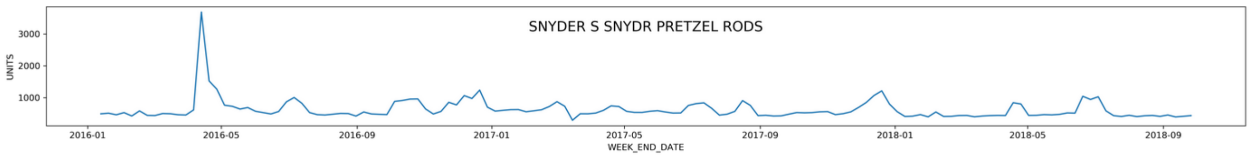
The graph shows the following:

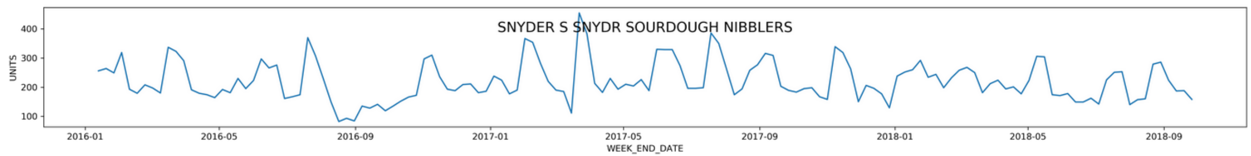
- The highest number is close to 80,000 and lowest is close to 20,000 units.
- There is no evident pattern or trend.
- The spikes can be seen in either direction and appear at no regular or constant interval.

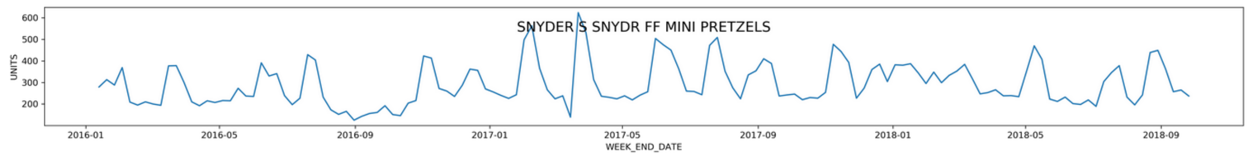
Units Sold Per Week at Product Level

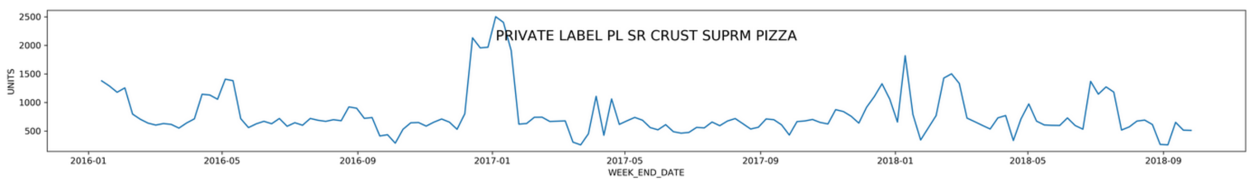
We will first look at product sales at the category level, looking for trends and seasonal patterns as before.

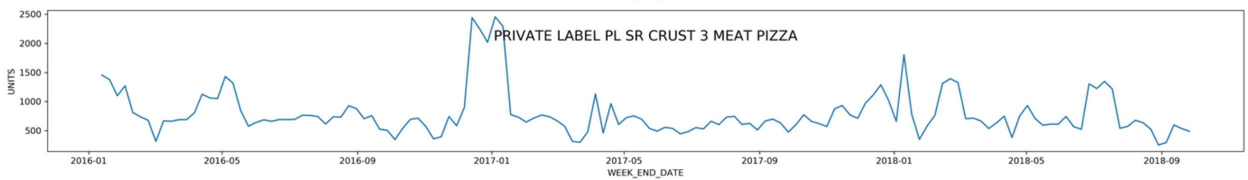


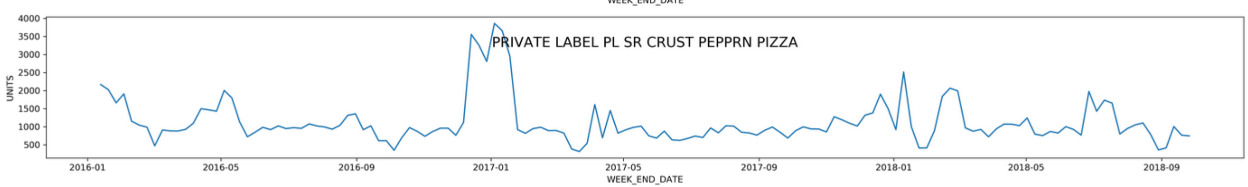

SNYDER S SNYDR PRETZEL RODS
UNITS
WEEK_END_DATE

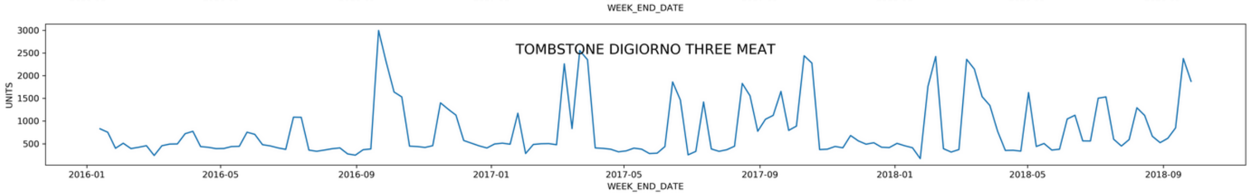

SNYDER S SNYDR SOURDOUGH NIBBLERS
UNITS
WEEK_END_DATE

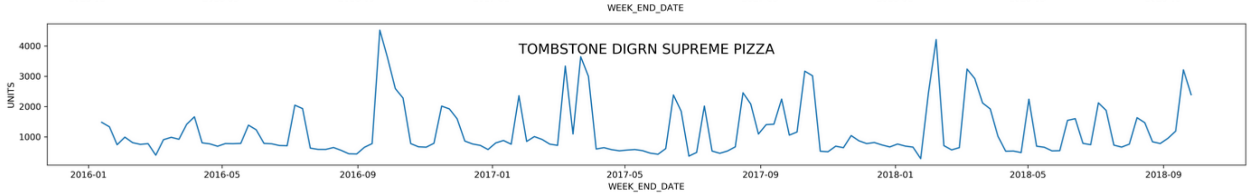

SNYDER S SNYDR FF MINI PRETZELS
UNITS
WEEK_END_DATE

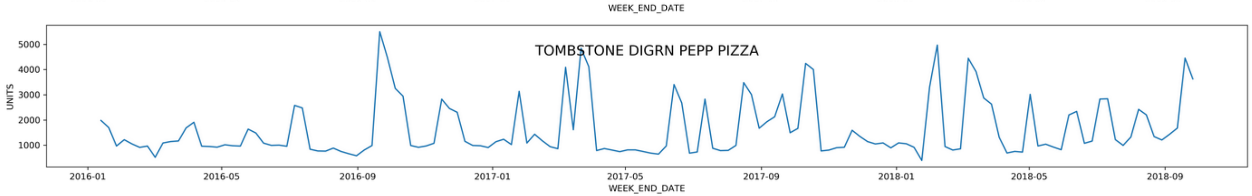

PRIVATE LABEL PL SR CRUST SUPRM PIZZA
UNITS
WEEK_END_DATE

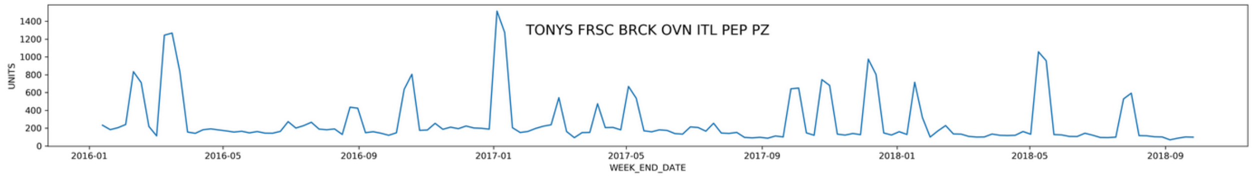

PRIVATE LABEL PL SR CRUST 3 MEAT PIZZA
UNITS
WEEK_END_DATE


PRIVATE LABEL PL SR CRUST PEPPRN PIZZA
UNITS
WEEK_END_DATE


TOMBSTONE DIGIORNO THREE MEAT
UNITS
WEEK_END_DATE


TOMBSTONE DIGRN SUPREME PIZZA
UNITS
WEEK_END_DATE


TOMBSTONE DIGRN PEPP PIZZA
UNITS
WEEK_END_DATE

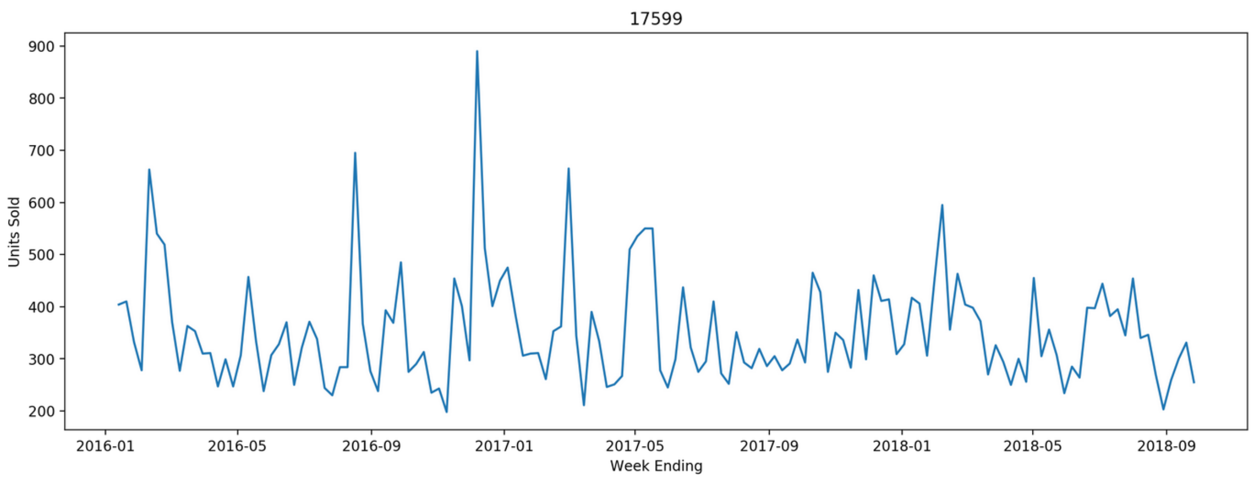


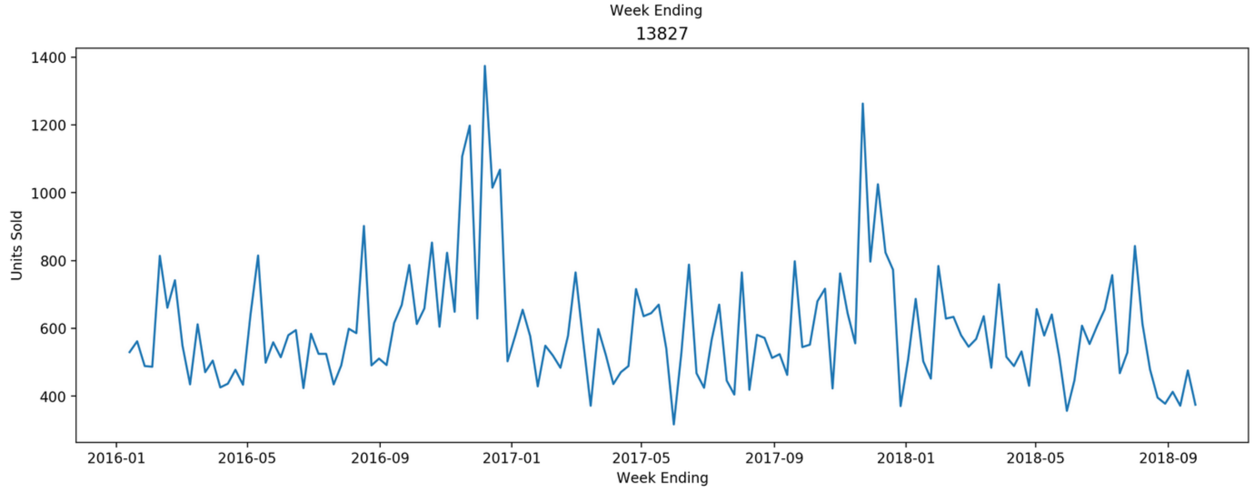
In reviewing the graphs for each product, we find:

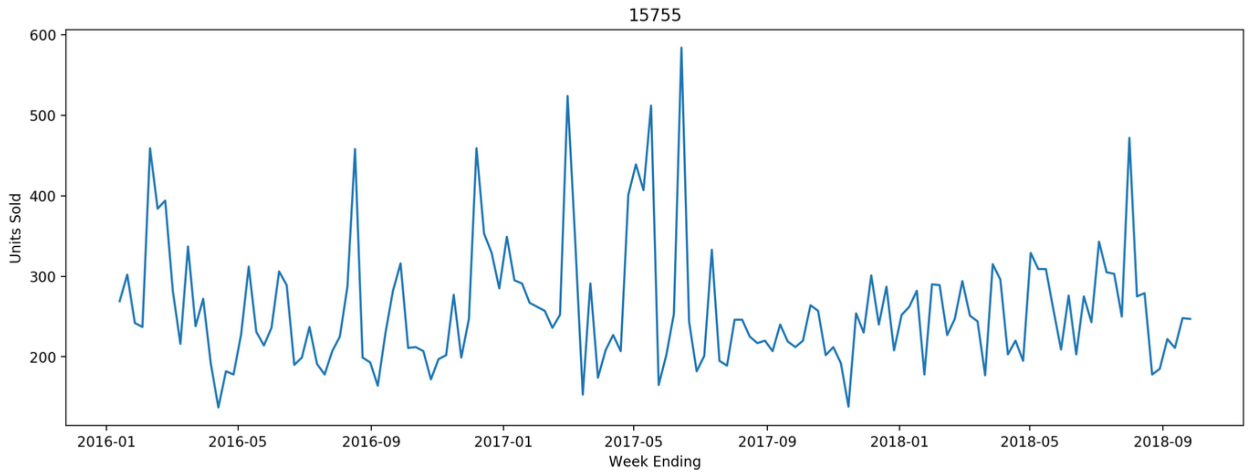
- No increasing/decreasing trends for the sale of products over time.
- No seasonal patterns seen on individual product sales.
- Products by same manufacturer have similar patterns (spikes and drops).

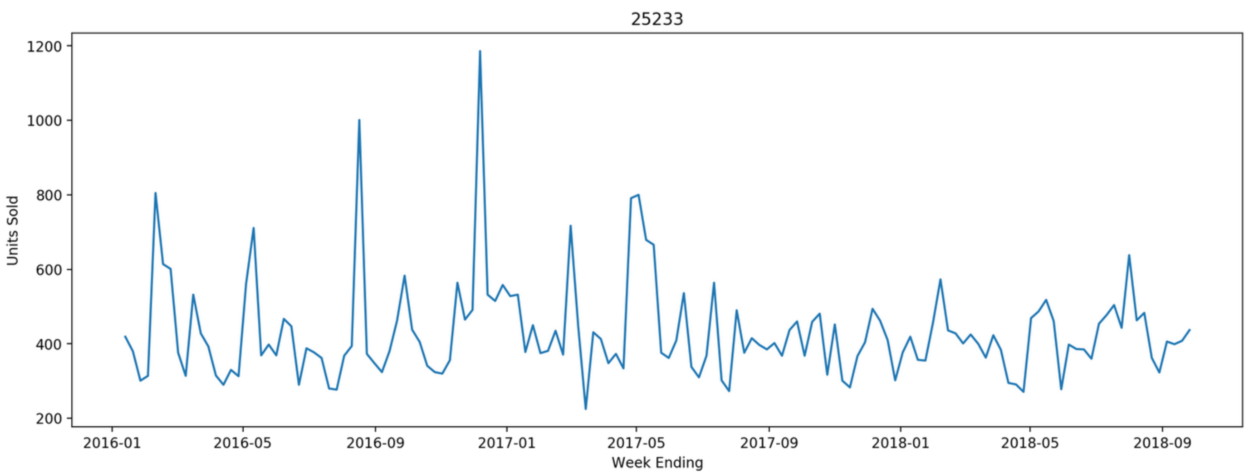
Units Sold Per Week at Store Level

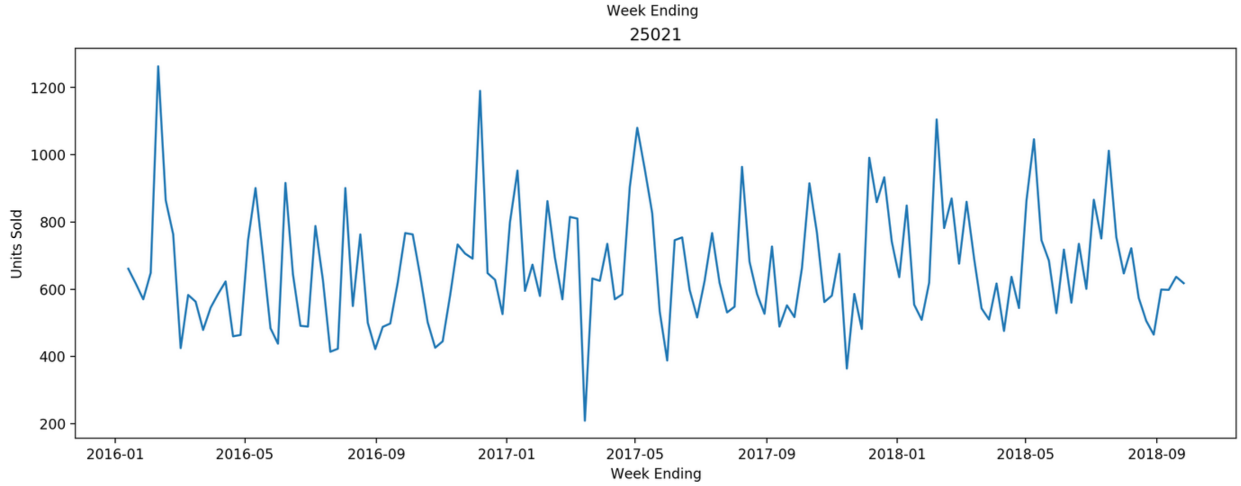
Next, we will look at store level demand patterns, again looking for trends and seasonal patterns. We will randomly look at 5 stores since we have not been given any information that would indicate that store demand would be uniquely different in any particular area.











For the randomly selected store numbers, we can see no trends or seasonal patterns in the graphs. The graphs were created for several more stores and the data showed no increasing or decreasing trends nor any seasonality.

Step 2 – Data Wrangling

Now that we understand the data a bit better, we are going to take a closer look and wrangle (or pre-process) the data in a way that will allow us to use it in a prediction model. This will involve:

- converting categorical features to a numeric representation
- making sure that there are no missing values in our categorical features (we already checked numeric ones)
- removing features that don't add value to the predication model, and
- choosing an encoding scheme by which to covert the categorical features to numeric features.

We will first look at the categorical features in each dataset.

Categorical Features

DATASET 1: 'SALES' Dataframe

The 'SALES' dataframe contains the following categorical features:

- STORE_NUM - store number (*key value)
- UPC - (Universal Product Code) product specific identifier (* key value)
- DISPLAY - product was a part of in-store promotional display
- FEATURE - product was in in-store circular

STORE_NUM and UPC codes are key variables and will only used to merge datasets. The other categorical variables are DISPLAY and FEATURE, both of which are already coded numerically ('1' or '0') for use in the model and do not need any wrangling.

DATASET 2: 'PRODUCTS' Dataframe

The 'PRODUCTS' dataframe contains the following features (all categorical features):

- UPC - (Universal Product Code) product specific identifier (* key value)
- DESCRIPTION - product description
- MANUFACTURER - product manufacturer/supplier
- CATEGORY - product category
- SUB_CATEGORY - product sub-category
- PRODUCT_SIZE - package size/quantity

STORE_NUM and UPC codes are key variables and will only used to merge datasets. 'DESCRIPTION' contains information already available in 'CATEGORY', 'SUB_CATEGORY' and 'PRODUCT_SIZE' and can be dropped.

'MANUFACTURER', 'CATEGORY' and 'SUB_CATEGORY' do not have an order or sequence. Thus, they will be converted to numerical using 'One Hot Encoder'.

The 'PRODUCT_SIZE' feature has different unit sizes for each category. As such, we will create numerical bins for this to be relevant in the model.

DATASET 3: 'STORES' Dataframe

The following categorical variables are in the STORES dataframe:

- STORE_ID - store number
- STORE_NAME - Name of store
- ADDRESS_CITY_NAME - city
- ADDRESS_STATE_PROV_CODE - state

- MSA_CODE - (Metropolitan Statistical Area) Based on geographic region and population density
- SEG_VALUE_NAME - Store Segment Name

STORE_ID – Same as before—key value for merging dataframes.

STORE_NAME & ADDRESS_CITY_NAME – Since there are 72/51 unique names out of 76 different unique stores, we will drop these features due to high cardinality.

ADDRESS_STATE_PROV_CODE and MSA_CODE - Again, there is no order in these categories, so we will use One Hot Encoder on these variables.

SEG_VALUE_NAME—Store segments are divided into 3 categories: upscale, mainstream and value. Upscale stores are just what they sound like; they are normally located in high income neighborhoods and offer more high-end products. Mainstream is middle of the road, mostly located in middle class areas, offering a mix of upscale and value product. Value stores cater more to low income customers, so there will be more focus on low prices than anything else. Since there is some type of order, we will map VALUE as '1', MAINSTREAM as '2' and UPSCALE AS '3'.

Continuous Features

DATASET 1: 'SALES' Dataframe

The 'SALES' dataframe contains the following continuous features:

- BASE_PRICE - base price of item
- UNITS - units sold (target)

```
# Check the null values for the numerical features.
sales[[ 'BASE_PRICE', 'UNITS']].isna().sum()
```

```
BASE_PRICE    12
UNITS          0
dtype: int64
```

Imputing the missing values in the Base Price

In looking for null values, we found that BASE_PRICE has 12. This was filled by computing the average price for the same product as found in other stores.

```
# Create a new dataframe which will have "average base price" for the combination of STORE_NUM and UPC.
# Will be used to impute missing values.
avg_price = sales.groupby(['STORE_NUM', 'UPC'])['BASE_PRICE'].mean().reset_index()
```

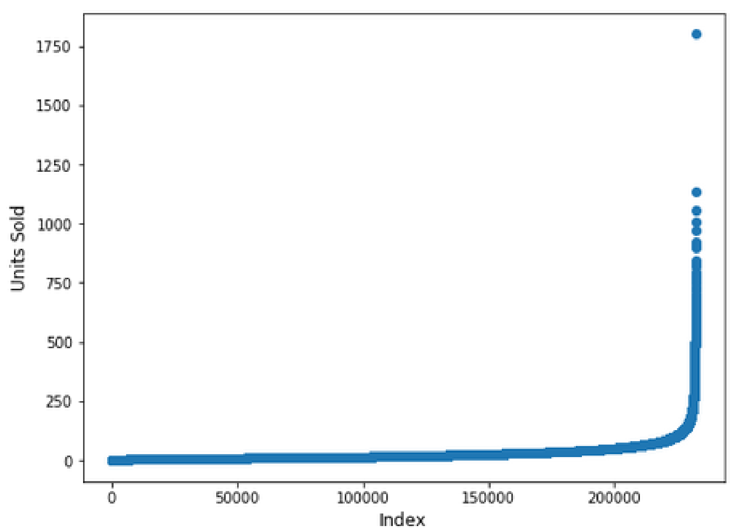
```
avg_price
```

	STORE_NUM	UPC	BASE_PRICE
0	367	1111009477	1.489859
1	367	1111009497	1.490634
2	367	1111085319	1.843451
3	367	1111085345	1.827183
4	367	1111085350	2.322113
...	...	...	...

Also, UNITS does not have any missing values, but it does have 21 outliers as shown in the following plot:

```
# Scatter plot for UNITS variable, sorted by target variable and scatter plot to see if there are outliers.
%matplotlib inline

plt.figure(figsize=(8,6))
plt.scatter(x = range(sales.shape[0]), y = np.sort(sales['UNITS'].values))
plt.xlabel('Index', fontsize=12)
plt.ylabel('Units Sold', fontsize=12)
plt.show()
```




```
# Number of data points where units are more than 750
sales['UNITS'][sales.UNITS > 750].shape[0]
```

```
21
```

This is a relatively small number of outliers compared to the overall dataset so they were removed. We are left with 232,287 rows of data.

DATASET 3: 'STORES' Dataframe

The STORES dataframe has the following continuous features:

- PARKING_SPACE_QTY
- SALES_AREA_SIZE_NUM
- AVG_WEEKLY_BASKETS

In checking for null values, we found 51 in the PARKING_SPACE_QTY variable. Since it is reasonable to assume that the number of parking spaces would be somewhat related to the store size, we checked the correlation between PARKING_SPACE_QTY and SALES_AREA_SIZE_NUM.

```
# Check correlation
stores[['PARKING_SPACE_QTY','SALES_AREA_SIZE_NUM']].corr()
```

	PARKING_SPACE_QTY	SALES_AREA_SIZE_NUM
PARKING_SPACE_QTY	1.000000	0.763274
SALES_AREA_SIZE_NUM	0.763274	1.000000

Since the correlation of PARKING_SPACE_QTY with SALES_AREA_SIZE_NUM is high, we can drop PARKING_SPACE_QTY as it will not add much value to the model.

Step 3 – Baseline Model Development

The next step in the project is to develop a baseline model. To do this, the data will be split into five different train/validation datasets and tested using 5 different algorithms.

Validation Strategy

Prior to doing baseline model development, a discussion of the evaluation metrics that were considered are in order. Since the problem we are analyzing is demand forecast, there are two possible problems in store operations—too much product, resulting in higher than necessary stocking costs, or not enough product, resulting in lost sales from empty shelf space when customers want to purchase a product. In business terms, lost revenue hurts the business (has a greater negative effect on profit) than too much product on the shelf.

There are three possible evaluation metrics for measuring this "error"—mean absolute error, the mean of the difference between forecast and actual; root mean squared error, the mean of the difference squared; and root mean log squared error, the mean of the difference in the log of the forecast and actual squared. To summarize, the first two methods essentially treat the differences between forecasted overages and shortages the same, which in practical terms, is not best for our model. We could minimize the error in the forecasted values, but it would not minimize the effect on profit. The third method, root mean log squared error, reduces the error on overstocking over understocking. As such, we will use this metric, RMLSE, in the model to predict demand.

Baseline Model Establishment

Now that we have the metric for evaluating our model, we will establish a baseline model using basic modeling techniques. After this is done, we can use more advance techniques to improve performance of the model. This will enable us to spot problems or bugs in these models, as any score which is below our baseline model is not good enough.

So for the baseline, we will attempt to predict demand using the mean demand from historical data for a particular store and product using Simple Moving Average. We will train the data on linear Regression based models and Decision Tree models.

The steps for this process will consist of the following:

- Merge datasets from Step 2
- Create train and validation sets
- Perform Mean Prediction using different models
- Evaluate results
- Select model for further development

Merge Datasets from Step 2

Here is a record from the dataset after merging SALES, PRODUCTS and STORES:

```
basemodel.loc[0]
```

```
WEEK_END_DATE          14-Jan-09
STORE_NUM                    367
UPC                   1111009477
BASE_PRICE                  1.57
FEATURE                        0
DISPLAY                        0
UNITS                         13
MANUFACTURER_1                 1
MANUFACTURER_2                 0
MANUFACTURER_3                 0
MANUFACTURER_4                 0
MANUFACTURER_5                 0
MANUFACTURER_6                 0
MANUFACTURER_7                 0
MANUFACTURER_8                 0
MANUFACTURER_9                 0
CATEGORY_1                     1
CATEGORY_2                     0
CATEGORY_3                     0
CATEGORY_4                     0
SUB_CATEGORY_1                 1
SUB_CATEGORY_2                 0
SUB_CATEGORY_3                 0
SUB_CATEGORY_4                 0
SUB_CATEGORY_5                 0
SUB_CATEGORY_6                 0
SUB_CATEGORY_7                 0
PRODUCT_SIZE                   2
ADDRESS_STATE_PROV_CODE_1      1
ADDRESS_STATE_PROV_CODE_2      0
ADDRESS_STATE_PROV_CODE_3      0
ADDRESS_STATE_PROV_CODE_4      0
MSA_CODE_1                     1
MSA_CODE_2                     0
MSA_CODE_3                     0
MSA_CODE_4                     0
MSA_CODE_5                     0
MSA_CODE_6                     0
MSA_CODE_7                     0
MSA_CODE_8                     0
MSA_CODE_9                     0
SEG_VALUE_NAME                 1
SALES_AREA_SIZE_NUM        24721
AVG_WEEKLY_BASKETS         12707
Name: 0, dtype: object
```

We can see the preprocessed features have been successfully included.

Create Validation Sets

In order to create the validation sets, we must use a time-based split since our data is time series data. Also, since there is a one-week gap to accommodate manufacturer product deliver, we will keep this gap between the train and validation datasets. Lastly, by creating five different training and validation sets from our entire dataset will allow us to determine whether there is consistency across different points of time.

Basic Prediction Models

The five basic prediction models that we will use are:

- Regression Models
  - Basic mean prediction
  - Simple moving average
  - Linear regression
- Decision Tree Models
  - Basic Decision Tree
  - RandomForest

We will run these models on all five train/validation datasets and compare the mean scores of the results.

Basic Mean Prediction

```
RMSLE ON TRAINING SET:  1 :  0.5902468460088598
RMSLE ON VALIDATION SET:  1 :  0.5887816704436897
=====================================================================
RMSLE ON TRAINING SET:  2 :  0.591251579931832
RMSLE ON VALIDATION SET:  2 :  0.6263156060802706
=====================================================================
RMSLE ON TRAINING SET:  3 :  0.5917841764867795
RMSLE ON VALIDATION SET:  3 :  0.47837118281730495
=====================================================================
RMSLE ON TRAINING SET:  4 :  0.5914233373769653
RMSLE ON VALIDATION SET:  4 :  0.5811759211472836
=====================================================================
RMSLE ON TRAINING SET:  5 :  0.5916269229162222
RMSLE ON VALIDATION SET:  5 :  0.718159952727328
=====================================================================
Mean RMSLE on Train:  0.5912665725441318
Mean RMSLE on Valid:  0.5985608666431753
```

This result is just ok. We need to try more models to see how it compares.

Simple Moving Average

```
RMSLE ON TRAINING SET:  1 :  0.532290520757075
RMSLE ON VALIDATION SET:  1 :  0.5469206496913668
=====================================================================
RMSLE ON TRAINING SET:  2 :  0.5332313430963387
RMSLE ON VALIDATION SET:  2 :  0.6421015703332319
=====================================================================
RMSLE ON TRAINING SET:  3 :  0.5335581839226429
RMSLE ON VALIDATION SET:  3 :  0.46149085909723564
=====================================================================
RMSLE ON TRAINING SET:  4 :  0.5324765840327685
RMSLE ON VALIDATION SET:  4 :  0.5878031103068386
=====================================================================
RMSLE ON TRAINING SET:  5 :  0.5318645623862213
RMSLE ON VALIDATION SET:  5 :  0.7558881602487321
=====================================================================
Mean RMSLE on Train:  0.5326842388390093
Mean RMSLE on Valid:  0.598840869935481
```

This model didn't perform much better than the Mean Prediction model.

Linear Regression

```
RMSLE ON TRAINING SET:  1 :  0.9913600210472604
RMSLE ON VALIDATION SET:  1 :  0.9149383085760163
=====================================================================
RMSLE ON TRAINING SET:  2 :  0.9959413624250854
RMSLE ON VALIDATION SET:  2 :  0.9020762359721142
=====================================================================
RMSLE ON TRAINING SET:  3 :  0.9933369914739013
RMSLE ON VALIDATION SET:  3 :  0.9648215340740681
=====================================================================
RMSLE ON TRAINING SET:  4 :  0.9976746323190893
RMSLE ON VALIDATION SET:  4 :  0.9600735777560083
=====================================================================
RMSLE ON TRAINING SET:  5 :  0.9933891252655116
RMSLE ON VALIDATION SET:  5 :  0.9743894218135638
=====================================================================
Mean RMSLE on Train:  0.9943404265061696
Mean RMSLE on Valid:  0.9432598156383541
```

Linear regression performed much worse than the other two baseline models.

Basic Decision Tree

```
RMSLE ON TRAINING SET:  1 :  0.41666120991123284
RMSLE ON VALIDATION SET:  1 :  0.455256744737524
====================================================================
RMSLE ON TRAINING SET:  2 :  0.41662543761463344
RMSLE ON VALIDATION SET:  2 :  0.4938950849210863
====================================================================
RMSLE ON TRAINING SET:  3 :  0.41634106921512387
RMSLE ON VALIDATION SET:  3 :  0.460788647437876
====================================================================
RMSLE ON TRAINING SET:  4 :  0.4159105014564439
RMSLE ON VALIDATION SET:  4 :  0.5179308424873471
====================================================================
RMSLE ON TRAINING SET:  5 :  0.41584465251212777
RMSLE ON VALIDATION SET:  5 :  0.5894093982947831
====================================================================
Mean RMSLE on Train:  0.4162765741419124
Mean RMSLE on Valid:  0.5034561435757233
```

Decision Tree performed much better than any of the previous linear baseline models.

RandomForest

```
RMSLE ON TRAINING SET:  1 :  0.4237387425772417
RMSLE ON VALIDATION SET:  1 :  0.4408882813939116
====================================================================
RMSLE ON TRAINING SET:  2 :  0.42370580798531027
RMSLE ON VALIDATION SET:  2 :  0.4714789603088615
====================================================================
RMSLE ON TRAINING SET:  3 :  0.42347475661984085
RMSLE ON VALIDATION SET:  3 :  0.4464008515972421
====================================================================
RMSLE ON TRAINING SET:  4 :  0.4230657735424195
RMSLE ON VALIDATION SET:  4 :  0.5083174671925305
====================================================================
RMSLE ON TRAINING SET:  5 :  0.42302395207577204
RMSLE ON VALIDATION SET:  5 :  0.5721305092171302
====================================================================
Mean RMSLE on Train:  0.4234018065601169
Mean RMSLE on Valid:  0.4878432139419352
```

The RandomForest model seems to be a slight improvement over the Decision Tree model.

This will be the baseline model used for the remainder of the project.

Validation Strategy Revisited

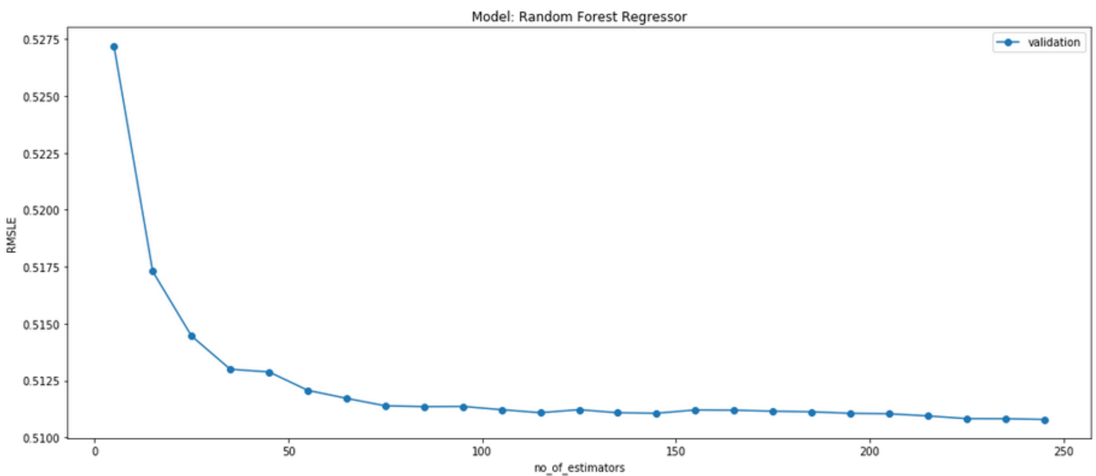
Step 4 – Optimize Model

Using the merged dataset and validation strategy from Step 3, we will perform the following in order to optimize our model:

- Feature Engineering
- Gradient Boosting & Hyperparameter Tuning
- Final Model Evaluation & Feature Importance

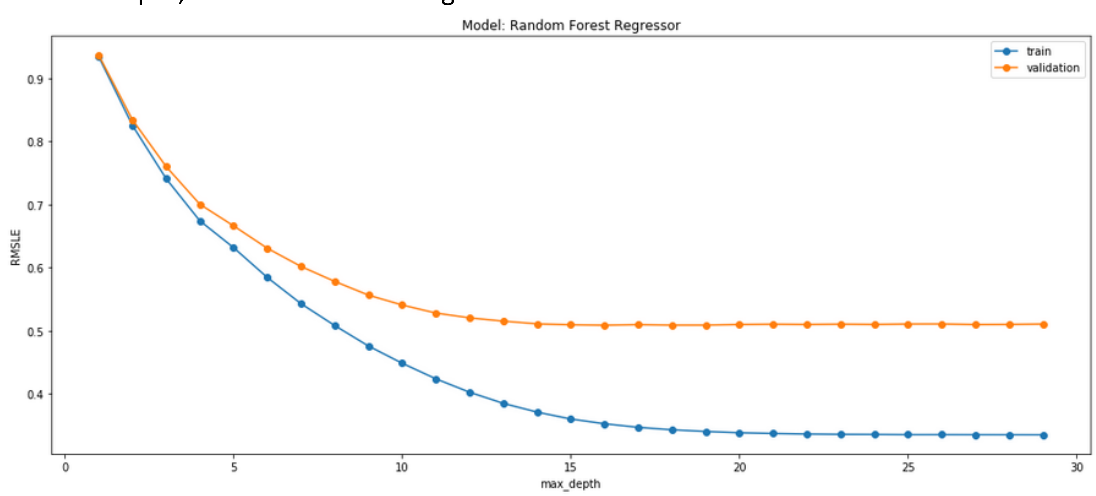
Feature Engineering

So first we will calculate the performance of the model using the default features and then we will try to tune the parameters to optimize results.

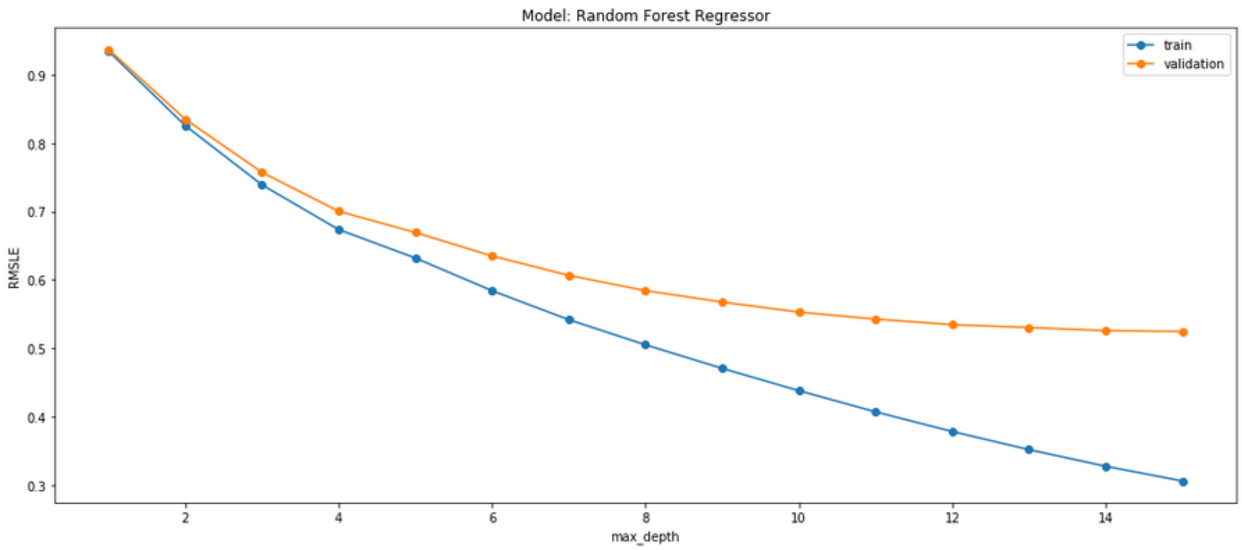


Running the RandomForest regression model using the number of estimators ranging from 5 to 245 in increments of 10 produced the lowest RMSLE around 175 estimators and seemed pretty consistent after that.

For Max Depth, we have the following results:



Add Time-Based Features



So, we can still see that the value of RMSLE on both train and validation set is getting somewhat stable after max depth 10. Now, we will keep on adding the new features to the data and check if it is improving the results or not.

Add New Features

For each STORE_ID we will find:

- Unique number of MANUFACTURERS
  - For each of the stores we will find out the number of unique manufactures as a feature.
  - We are assuming that more manufactures will give more options to the customers and will impact sales.
  - We will have to use the original sales and product data to calculate this as we have encoded this feature during the pre-processing step.
- Unique number of CATEGORY and SUB_CATEGORIES each store has

```
print('RMSLE on train set: ', rmsle_train)
print('RMSLE on validation set:', rmsle_valid)
```

```
RMSLE on train set:  0.4354111678465658
RMSLE on validation set: 0.5511201466160403
```

After doing this, we see a significant improvement in model performance, with the RMSLE on the validation dataset at 0.551.

Create Lag Feature

Now, we will create the lag features, which will be the number of units ordered of the same product from the same store at exactly one year ago.

```
# mean RMSLE on train and validation set
print('RMSLE on train set: ', rmsle_train)
print('RMSLE on validation set:', rmsle_valid)
```

```
RMSLE on train set:  0.4120974817815601
RMSLE on validation set: 0.5712462005883503
```

This time, we do not see much of a change in model performance, with the RMSLE on the validation dataset at 0.571.

New Feature – Previous Week's Price Difference

```
print('RMSLE on train set: ', rmsle_train)
print('RMSLE on validation set:', rmsle_valid)
```

```
RMSLE on train set:  0.41279920755574884
RMSLE on validation set: 0.5709107580081936
```

No change in model performance.

New Feature - Average Units Sold 2 Months Earlier

```
print('RMSLE on train set: ', rmsle_train)
print('RMSLE on validation set:', rmsle_valid)
```

```
RMSLE on train set:  0.3448230185588829
RMSLE on validation set: 0.4626667070555978
```

This feature has resulted in a significant improvement in model performance, lowering the RMSLE score on the validation dataset to 0.463.

Final Model and Feature Performance

From Step 3, we determined that we achieved the best results from 2 months of data in 14 different validation datasets. For our final model, we produced the following:

```
validation_df(data, week, no_of_months=2, no_of_validation = 14)
```

	train_start_1	train_end_1	train_start_2	validate_week	test_week	no_days_train_1	no_days_train_2	set_no
0	2011-07-13	2011-08-31	2011-07-27	2011-09-14	2011-09-28	56 days	56 days	set1
1	2011-07-06	2011-08-24	2011-07-20	2011-09-07	2011-09-21	56 days	56 days	set2
2	2011-06-29	2011-08-17	2011-07-13	2011-08-31	2011-09-14	56 days	56 days	set3
3	2011-06-22	2011-08-10	2011-07-06	2011-08-24	2011-09-07	56 days	56 days	set4
4	2011-06-15	2011-08-03	2011-06-29	2011-08-17	2011-08-31	56 days	56 days	set5
5	2011-06-08	2011-07-27	2011-06-22	2011-08-10	2011-08-24	56 days	56 days	set6
6	2011-06-01	2011-07-20	2011-06-15	2011-08-03	2011-08-17	56 days	56 days	set7
7	2011-05-25	2011-07-13	2011-06-08	2011-07-27	2011-08-10	56 days	56 days	set8
8	2011-05-18	2011-07-06	2011-06-01	2011-07-20	2011-08-03	56 days	56 days	set9
9	2011-05-11	2011-06-29	2011-05-25	2011-07-13	2011-07-27	56 days	56 days	set10
10	2011-05-04	2011-06-22	2011-05-18	2011-07-06	2011-07-20	56 days	56 days	set11
11	2011-04-27	2011-06-15	2011-05-11	2011-06-29	2011-07-13	56 days	56 days	set12
12	2011-04-20	2011-06-08	2011-05-04	2011-06-22	2011-07-06	56 days	56 days	set13
13	2011-04-13	2011-06-01	2011-04-27	2011-06-15	2011-06-29	56 days	56 days	set14

The results of our RandomForest model from this were as follows: