Data Information

The data to be used in this project came as three seperate CSVs, found at: https://www.kaggle.com/datasets/manjeetsingh/retaildataset. The stores data sheet has information about store type and size for each of the 45 stores. The sales data has weekly sales numbers for each department in each store from Feb 5th, 2010 to Oct 26th, 2012. The features dataset has weekly non-sales economic information from Feb 5th, 2010 through Jul 26th, 2013. Since the sales dataset has a shorter timeframe, its timeframe will be the time used for modeling. It is also worth noting that the dates in the dataset follow %d/%m/%Y format.

Merging the Data

The two datasets have vastly different numbers of rows since one has each week at a one of thr 45 stores be a row. We will want to get all of our data into one dataframe, so some merging and wrangling is going to have to happen. Merging the store and sales sheet is an easy merge to make work. That result can be merged with the features sheet using both Store and Date as keys to maintain the date range of the data in the sales dataset and keep the correct number of rows.

In [2]:
```
import pandas as pd

# Load the datasets
store_sheet = pd.read_csv('stores data-set.csv')
features_sheet = pd.read_csv('Features data set.csv')
sales_sheet = pd.read_csv('sales data-set.csv')
```

In [3]:
```
merge1 = sales_sheet.merge(store_sheet, on='Store', validate='m:1')
```

In [4]:
```
merge2 = merge1.merge(features_sheet, how='left', on=['Store', 'Date'],
                      validate='m:m')
```

In [5]:
```
data = merge2.drop(columns=['IsHoliday_y'])
data.columns = ['Store', 'Dept', 'Date', 'Weekly_Sales', 'IsHoliday', 'Store_Type',
                'Store_Size', 'Temperature', 'Fuel_Price', 'MarkDown1',
                'MarkDown2', 'MarkDown3', 'MarkDown4', 'MarkDown5', 'CPI',
                'Unemployment']
```

Data Preparation/Cleanup

With the datasets now merged, we need to clean up the data. The dates in the Date column are using a dd/mm/YYYY format, so that can be fixed.

There is also a lot of NaN data in the MarkDown fields since only dates after November 2011 have data. Without other information, selecting 0 or any other value (mean, median, etc) would fill in a large amount of space for each store since actual data is only available after November 2011. These columns contain "anonymized data related to promotional markdowns". It is unknown if these columns contain the sum of the value of products sold that were on sale that week from that department. It may be wise to remove the columns instead of trying to fill them with mostly similar data that would end up influencing the model for more time than there is actual data for.

The IsHoliday column is currently populated with true or false for its values. We will want to convert these to 1s and 0s.

Lastly, the store and department fields have numerical values but are categorical variables. We can convert those fields to strings to make life easier for ourselves.

A set of helper functions can be written to address concerns over the dates in the data in order to map the functions to individual series using lambda functions.

In [7]:
```
from datetime import datetime

# Writing a function to convert the dates
def date_converter(date_string):
    date_format = '%d/%m/%Y'
    date = datetime.strptime(date_string, date_format)
    return date


# Writing a function to convert the Boolean values
def bool_converter(bool):
    if bool == True:
        value = 1
    elif bool == False:
        value = 0
    else:
        value = 'Oops'
    return value


# Writing a function to convert values into strings
def string_converter(not_string):
    string = str(not_string)
    return string
```

In [8]:
```
# Converting the dates
data.Date = data.Date.map(lambda x: date_converter(x))

# Converting the T/F Values
data.IsHoliday = data.IsHoliday.map(lambda x: bool_converter(x))
```

Exploring Store Types

There are 3 store types, so finding some basic information about each store type would be wise. We may find it unwise to use the same model to make projections for all three store types.

In [10]:
```
data.groupby('Store_Type').agg({'Weekly_Sales' : 'mean', 'Store_Size' : 'mean'})
```

Out[10]:

Store_Type	Weekly_Sales	Store_Size
A	20099.568043	182231.285486
B	12237.075977	101818.735827
C	9519.532538	40535.725286

In [11]:
```
data.groupby('Store_Type').agg({'Weekly_Sales' : 'median', 'Store_Size' : 'median'}
```

Out[11]:

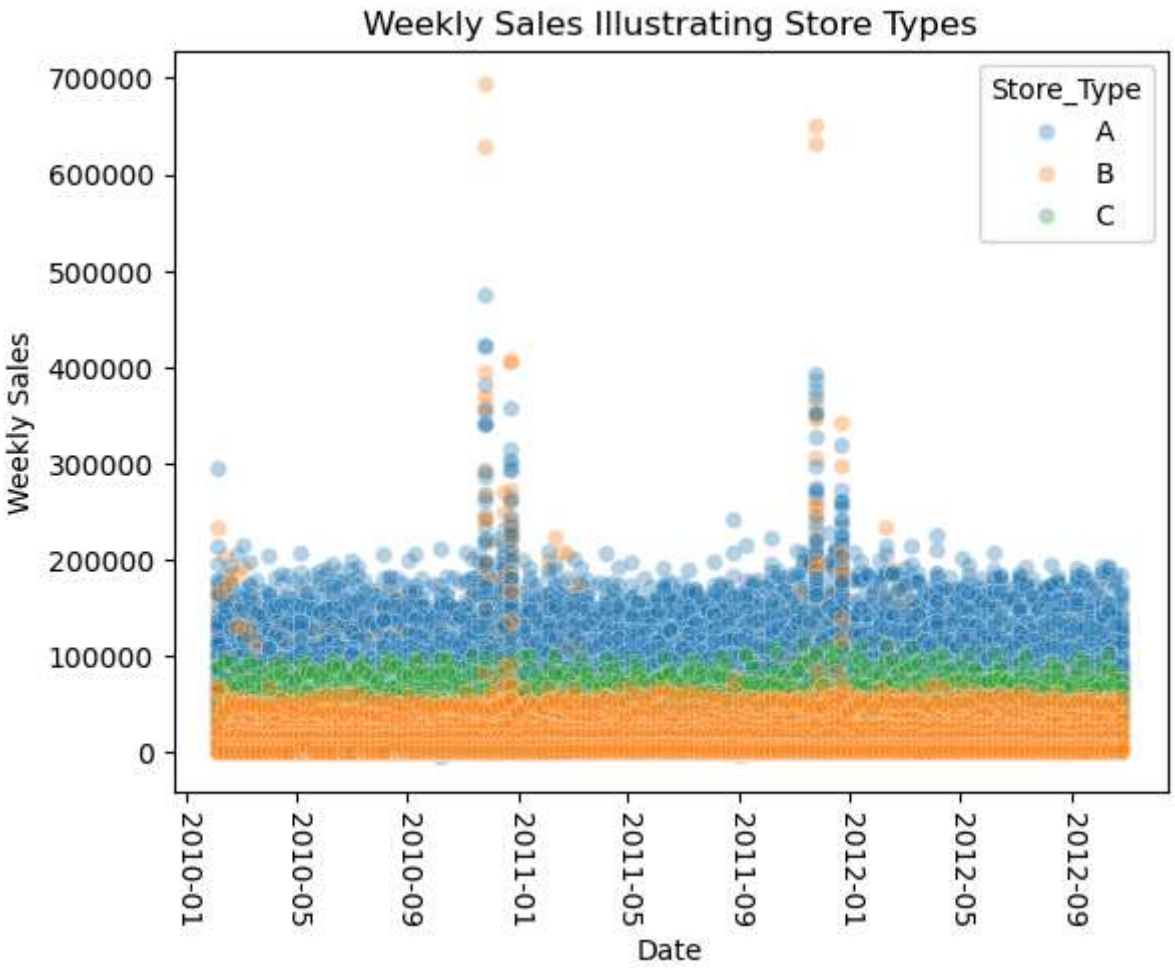
Store_Type	Weekly_Sales	Store_Size
A	10105.17	202505.0
B	6187.87	114533.0
C	1149.67	39910.0

We can see that the different store types have sizably different mean and median values for store size and weekly sales. The data should be split into 3 datasets, one for each store type. It would be better to have one model made for each store type.

By plotting the all of the weekly store sales and coloring by store type, we can illustrate this further to make sure we use separate models for each store type. We can split this data into three sets once we're done preparing the data

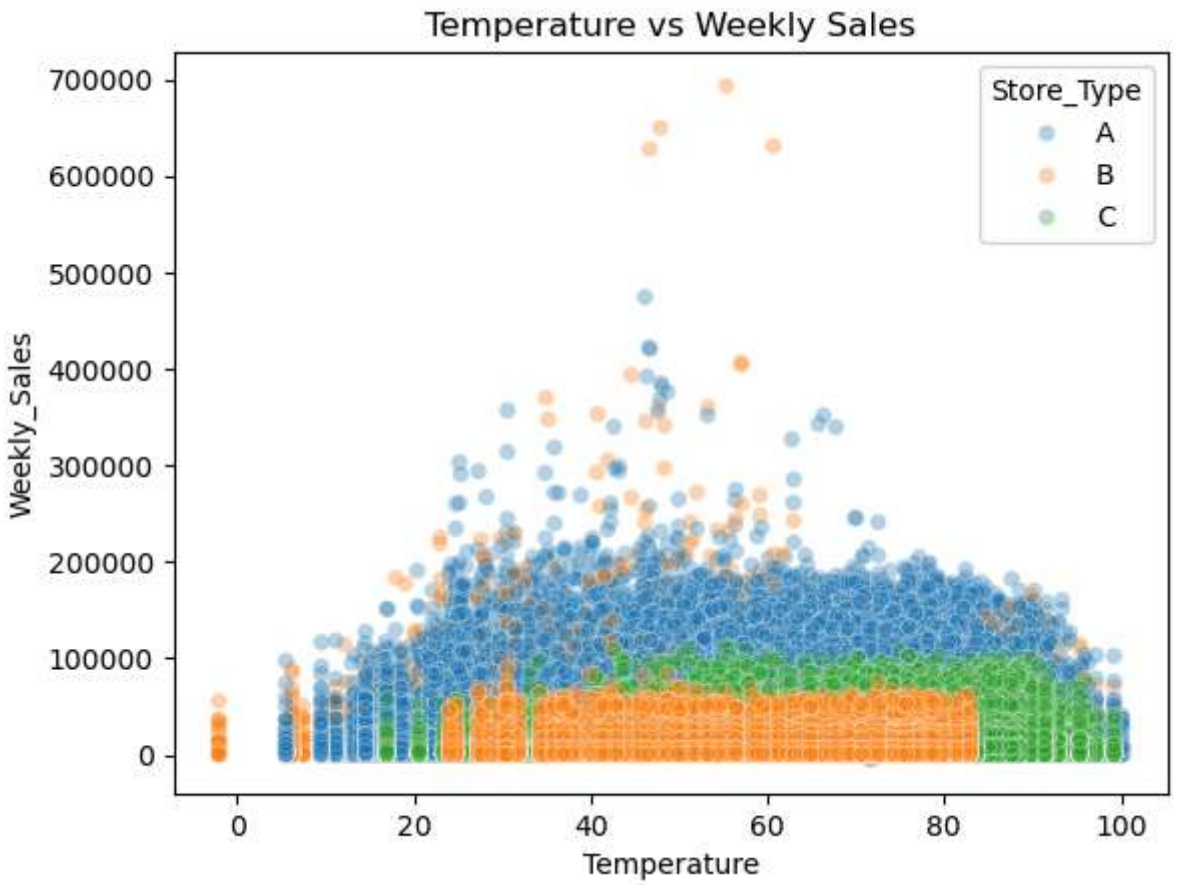
In [14]:
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.scatterplot(data=data, x='Date', y='Weekly_Sales', hue='Store_Type',
                alpha=0.33)
plt.ylabel('Weekly Sales')
plt.xticks(rotation=270)
plt.title('Weekly Sales Illustrating Store Types')
```

Out[14]:
```
Text(0.5, 1.0, 'Weekly Sales Illustrating Store Types')
```



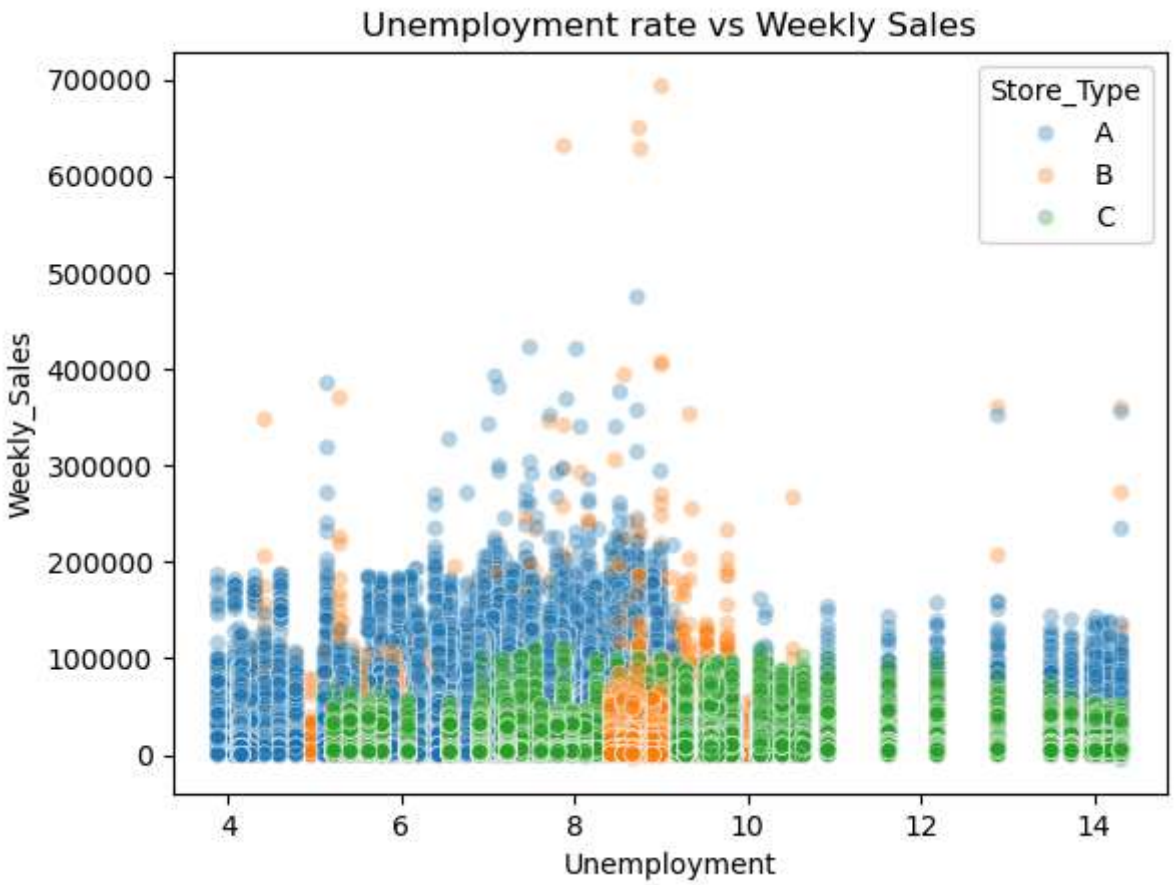
In [15]:
```
sns.scatterplot(data=data, x='Temperature', y='Weekly_Sales', hue='Store_Type',
                alpha=0.33)
plt.title('Temperature vs Weekly Sales')
```

Out[15]:
```
Text(0.5, 1.0, 'Temperature vs Weekly Sales')
```



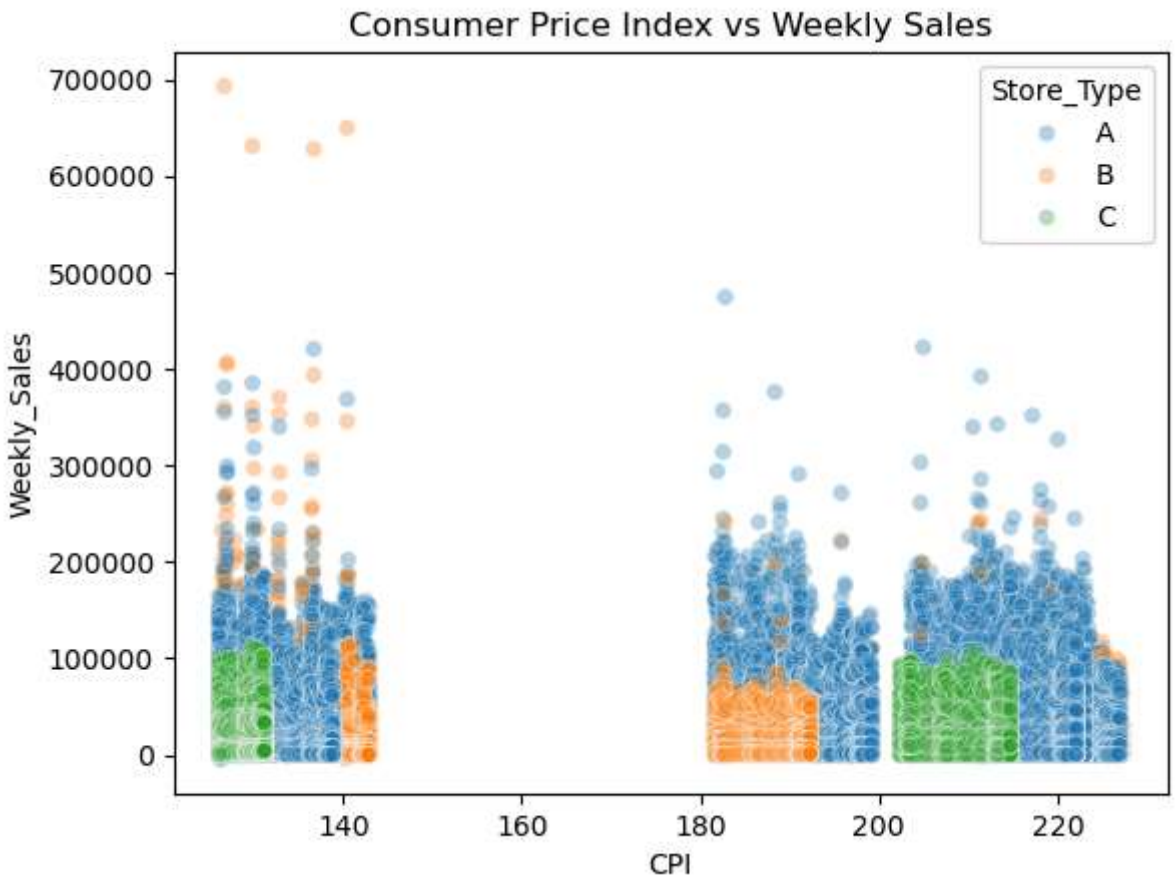
In [16]:
```
sns.scatterplot(data=data, x='Unemployment', y='Weekly_Sales', hue='Store_Type',
                alpha=0.33)
plt.title('Unemployment rate vs Weekly Sales')
```

Out[16]:
```
Text(0.5, 1.0, 'Unemployment rate vs Weekly Sales')
```



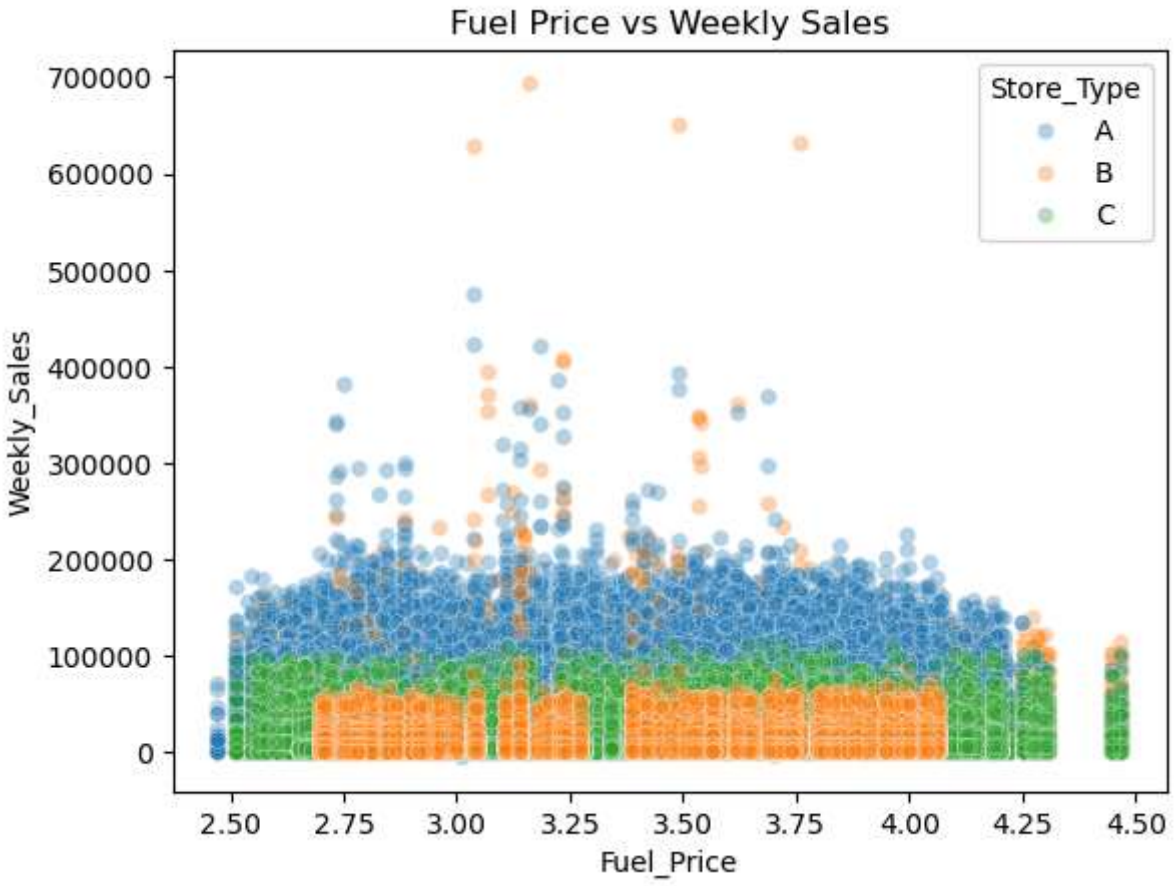
In [17]:
```
sns.scatterplot(data=data, x='CPI', y='Weekly_Sales', hue='Store_Type',
                alpha=0.33)
plt.title('Consumer Price Index vs Weekly Sales')
```

Out[17]:
```
Text(0.5, 1.0, 'Consumer Price Index vs Weekly Sales')
```



In [18]:
```
sns.scatterplot(data=data, x='Fuel_Price', y='Weekly_Sales', hue='Store_Type',
                lpha=0.33)
plt.title('Fuel Price vs Weekly Sales')
```

Out[18]:
```
Text(0.5, 1.0, 'Fuel Price vs Weekly Sales')
```



Addressing the MarkDown Columns

The five MarkDown columns contain anonymized sales data, but only for weeks after November 2011. This leaves a significant number of NA data that needs addressing.

In [20]:
```
md1missperc = sum(data.MarkDown1.isna()) / len(data.MarkDown1) * 100
md2missperc = sum(data.MarkDown2.isna()) / len(data.MarkDown2) * 100
md3missperc = sum(data.MarkDown3.isna()) / len(data.MarkDown3) * 100
md4missperc = sum(data.MarkDown4.isna()) / len(data.MarkDown4) * 100
md5missperc = sum(data.MarkDown5.isna()) / len(data.MarkDown5) * 100

print('The missing percentages are', md1missperc,
      md2missperc, md3missperc, md4missperc, md5missperc)
```

```
The missing percentages are 64.25718148824632 73.61102545247527 67.48084541120099 6
7.98467632896079 64.07903788220224
```

Each of these fields are more than half missing. Attempting to impute values for all of the NAs each of these fields would unduly affect the model no matter what value is chosen. So it will be easier to drop the five fields.

In [22]:
```
data = data.drop(columns=['MarkDown1', 'MarkDown2', 'MarkDown3',
                          'MarkDown4', 'MarkDown5'])
```

Splitting The Data by Store Type

The last thing to do before making dummy variables is going to be to split the data by store type via Boolean filtering.

In [24]:
```
dataA = data[data.Store_Type == 'A'].drop(columns=['Store_Type'])
dataB = data[data.Store_Type == 'B'].drop(columns=['Store_Type'])
dataC = data[data.Store_Type == 'C'].drop(columns=['Store_Type'])
```

Modeling with the Data

Models for time series data are specialized since the result is dependent on time itself and has the potential to be cyclical. Even still, seasonal data can provide other challenges. So to model this retail data, a SARIMA (Seasonal Autoregressive Integrated Moving Average) model will be a good fit for the training and testing data given the seasonal nature of the data.

SARIMA models can be generated from the statsmodels and pmdarima libraries. I plan to compare the results from the two models to determine which is better. One problem though is that for a SARIMA model to work as intended, it will require a single store and department combination. The current state of the datasets that are optimized for store type is that each week has multiple stores and departments of data. SARIMA wants consecutive data, not repetitive data. This means that one model can only fit one store and department combination. So the initial modeling will need to be scaled for that size. In a professional setting, this would be scaled up to fit a model for every store/department combination that has enough data.

Creating Training and Testing Sets

To create training and testing sets, we will want to index for a particular date. The end of the data comes in October 2012, so a simple model for this scenario (spans from 2/2010 to 10/2012 - 2.75 years) could make predictions for a six month span. Indexing for April 1, 2012 would be a suitable split then.

In [26]:
```
from sklearn.preprocessing import StandardScaler
standardizer = StandardScaler()

def scaler(df):
    scaled_df = pd.DataFrame()
    for col in df.columns():
        scaled_df[col] = standardizer(df[col])
    return scaled_df
```

In [27]:
```
# Splitting the Training and Testing Sets
trainA = dataA[dataA.Date < '2012-04-01'].sort_values('Date')
trainB = dataB[dataB.Date < '2012-04-01'].sort_values('Date')
trainC = dataC[dataC.Date < '2012-04-01'].sort_values('Date')

testA = dataA[dataA.Date >= '2012-04-01'].sort_values('Date')
testB = dataB[dataB.Date >= '2012-04-01'].sort_values('Date')
testC = dataC[dataC.Date >= '2012-04-01'].sort_values('Date')
```

With the training and testing split done, we want to determine which store and department should be used for each store type. Ideally, we would like to use the combination with the most data available. A quick function can provide a list with the best candidates store and departmenrt combinations.

In [29]:
```
def storedept_counter(df):
    dict = {}
    i = 0
    while i < len(df):
        combo = (df.Store.iloc[i], df.Dept.iloc[i])
        if combo in dict:
            dict[combo] += 1
        else:
            dict[combo] = 1
        i += 1
    list = []
    for key in dict:
        list.append((dict[key], key))
    list.sort(reverse=True)
    return list
```

In [30]:
```
As = storedept_counter(dataA)
Bs = storedept_counter(dataB)
Cs = storedept_counter(dataC)
```

In [31]:
```
print('A : ', As[0:5])
print('B : ', Bs[0:5])
print('C : ', Cs[0:5])
```

```
A :  [(143, (41, 98)), (143, (41, 97)), (143, (41, 96)), (143, (41, 95)), (143, (41,
94))]
B :  [(143, (45, 97)), (143, (45, 95)), (143, (45, 93)), (143, (45, 92)), (143, (45,
91))]
C :  [(143, (44, 98)), (143, (44, 97)), (143, (44, 96)), (143, (44, 95)), (143, (44,
94))]
```

It looks like most pairs have the full 143 rows of data. Type A will then use 1/1, B will use 3/1, and C will use 30/1. While these were not the ones at the top of the sort above, I did confirm that they each have the full 143 rows of data in excel. Now we can select the stores from the training and testing sets.

In [33]:
```
# Selecting the stores and departments in the training sets
trainA11 = trainA[(trainA['Store'] == 1) & (trainA['Dept'] == 1)]
```

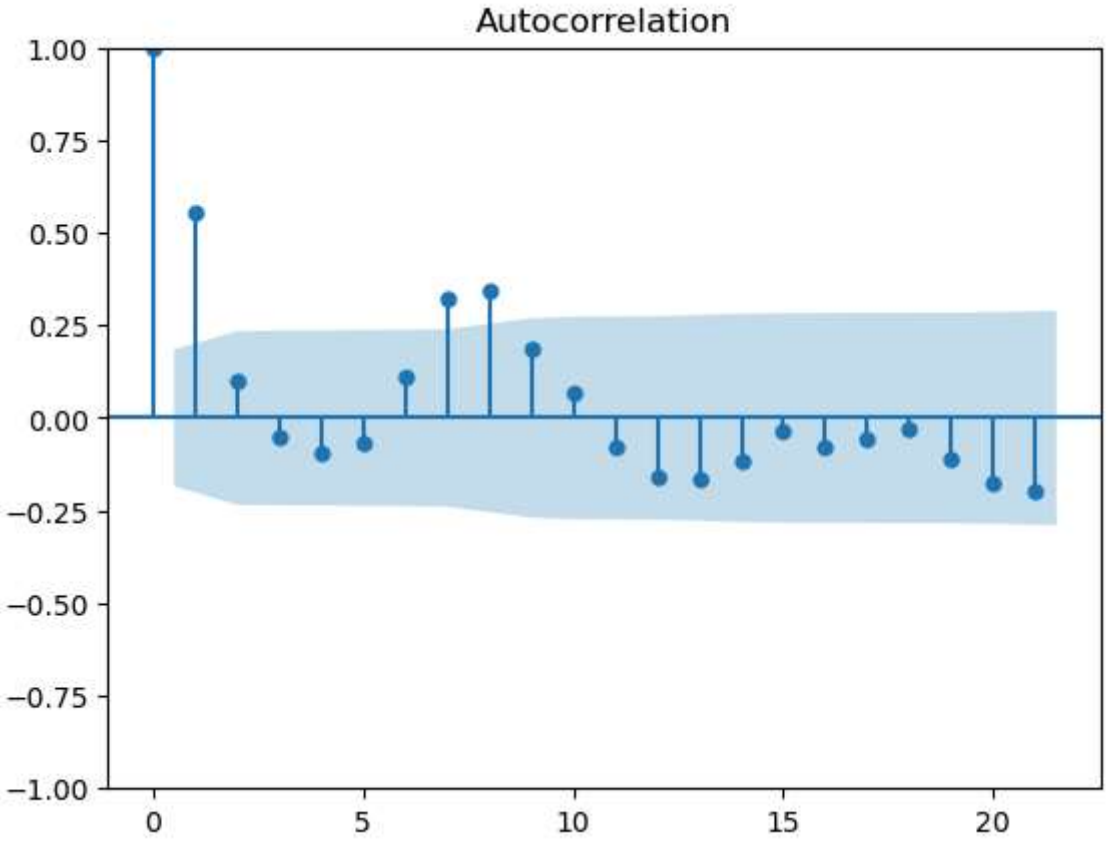
```
trainB31 = trainB[(trainB['Store'] == 3) & (trainB['Dept'] == 1)]
trainC301 = trainC[(trainC['Store'] == 30) & (trainC['Dept'] == 1)]

# Selecting the stores and departments in the testing sets
testA11 = testA[(testA['Store'] == 1) & (testA['Dept'] == 1)]
testB31 = testB[(testB['Store'] == 3) & (testB['Dept'] == 1)]
testC301 = testC[(testC['Store'] == 30) & (testC['Dept'] == 1)]
```

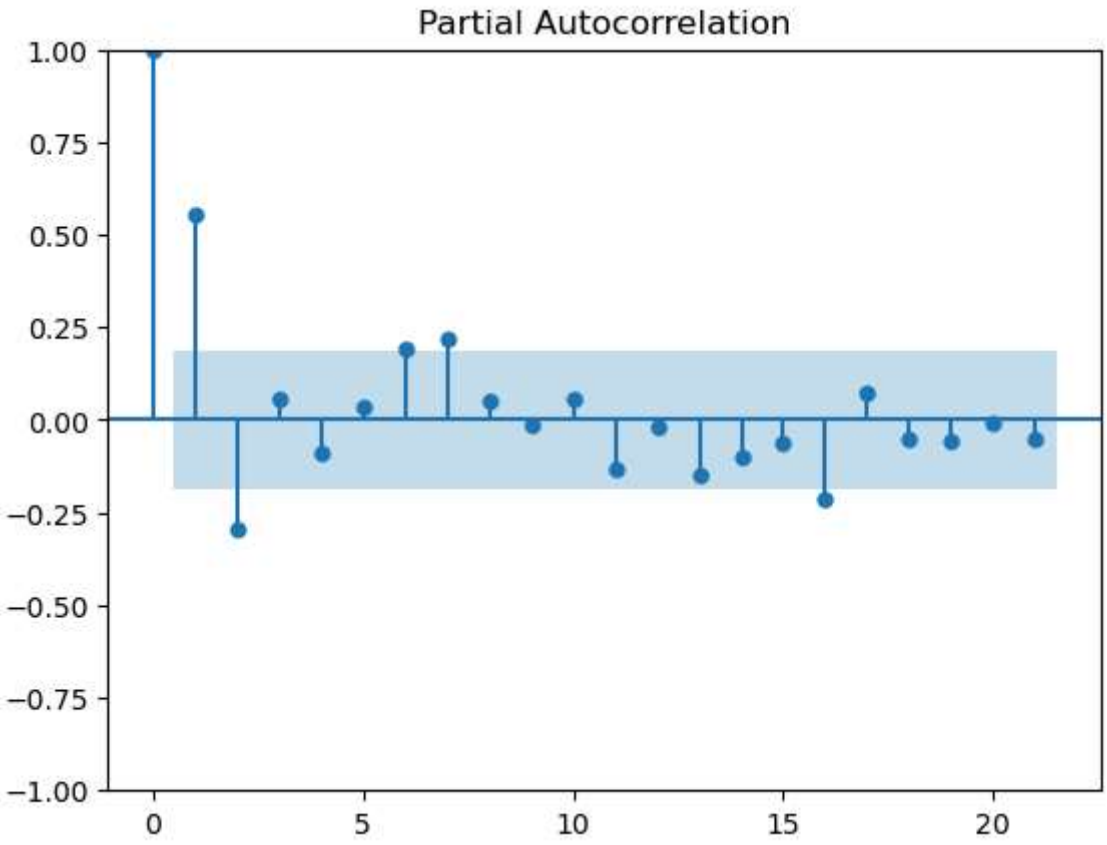
__ARIMA Model Parameters__ Since SARIMA models are just seasonal ARIMA models, we will need to figure out the ARIMA oriented parameters, p, d, and q. To fit those, we will want to use ACF and PACF plots from statsmodels. Endogeneous and Exogeneous variables are also necessary for model specification. So we will want to make those splits as well, which will include making sure the date is listed as the index value.

In [35]:
```
# Changing index for the training sets
trainA11.index = trainA11.Date
trainA11 = trainA11.drop(columns=['Date'])
trainB31.index = trainB31.Date
trainB31 = trainB31.drop(columns=['Date'])
trainC301.index = trainC301.Date
trainC301 = trainC301.drop(columns=['Date'])

# Changing index for the test sets
testA11.index = testA11.Date
testA11 = testA11.drop(columns=['Date'])
testB31.index = testB31.Date
testB31 = testB31.drop(columns=['Date'])
testC301.index = testC301.Date
testC301 = testC301.drop(columns=['Date'])
```

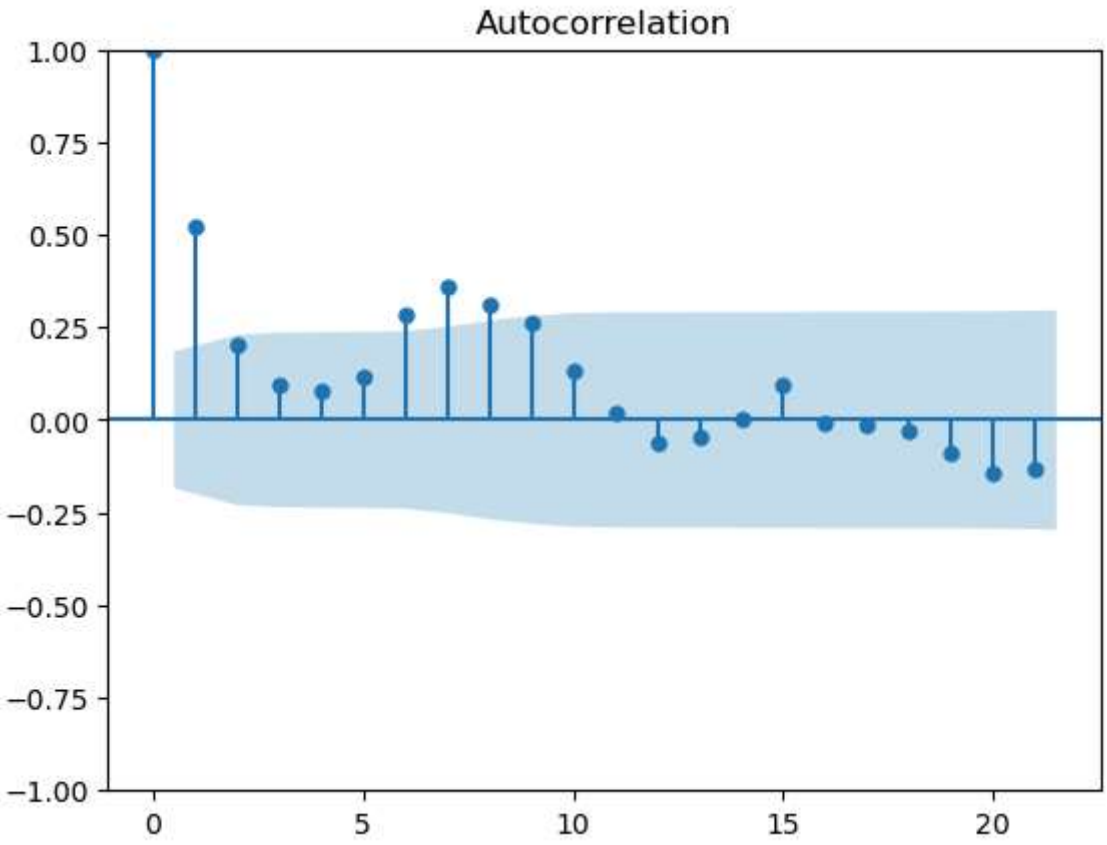
In [36]:
```
# Making Endo/Exo Split for Training Variables
trainA11_endo = trainA11['Weekly_Sales']
trainB31_endo = trainB31['Weekly_Sales']
trainC301_endo = trainC301['Weekly_Sales']

trainA11_exo = trainA11.drop(columns=['Store', 'Dept', 'Weekly_Sales',
                                      'Store_Size'])
trainB31_exo = trainB31.drop(columns=['Store', 'Dept', 'Weekly_Sales',
                                      'Store_Size'])
trainC301_exo = trainC301.drop(columns=['Store', 'Dept', 'Weekly_Sales',
                                        'Store_Size'])

# Making Endo/Exo Split for Testing Variables
testA11_endo = testA11['Weekly_Sales']
testB31_endo = testB31['Weekly_Sales']
testC301_endo = testC301['Weekly_Sales']

testA11_exo = testA11.drop(columns=['Store', 'Dept', 'Weekly_Sales',
                                    'Store_Size'])
testB31_exo = testB31.drop(columns=['Store', 'Dept', 'Weekly_Sales',
                                    'Store_Size'])
testC301_exo = testC301.drop(columns=['Store', 'Dept', 'Weekly_Sales',
                                      'Store_Size'])
```

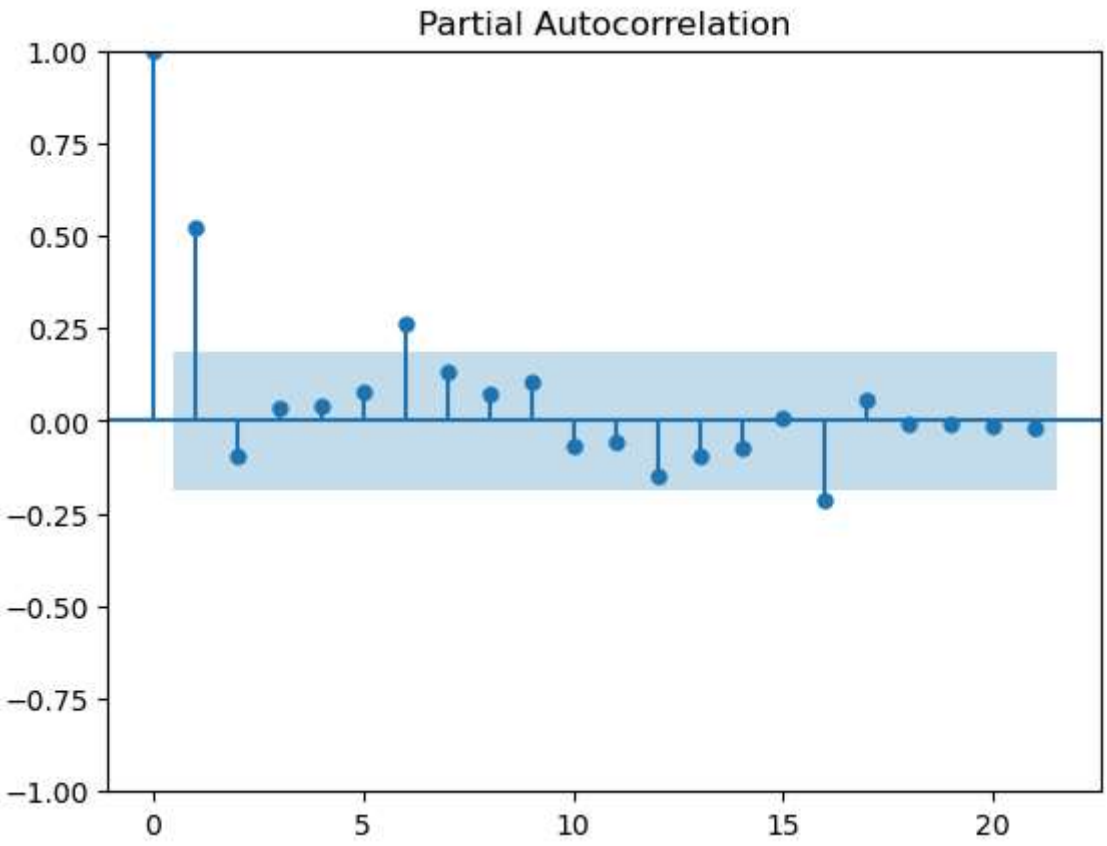
In [37]:
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Creating the A ACF plots
acfA = plot_acf(trainA11_endo)

# Creating the A PACF plots
pacfA = plot_pacf(trainA11_endo)
```





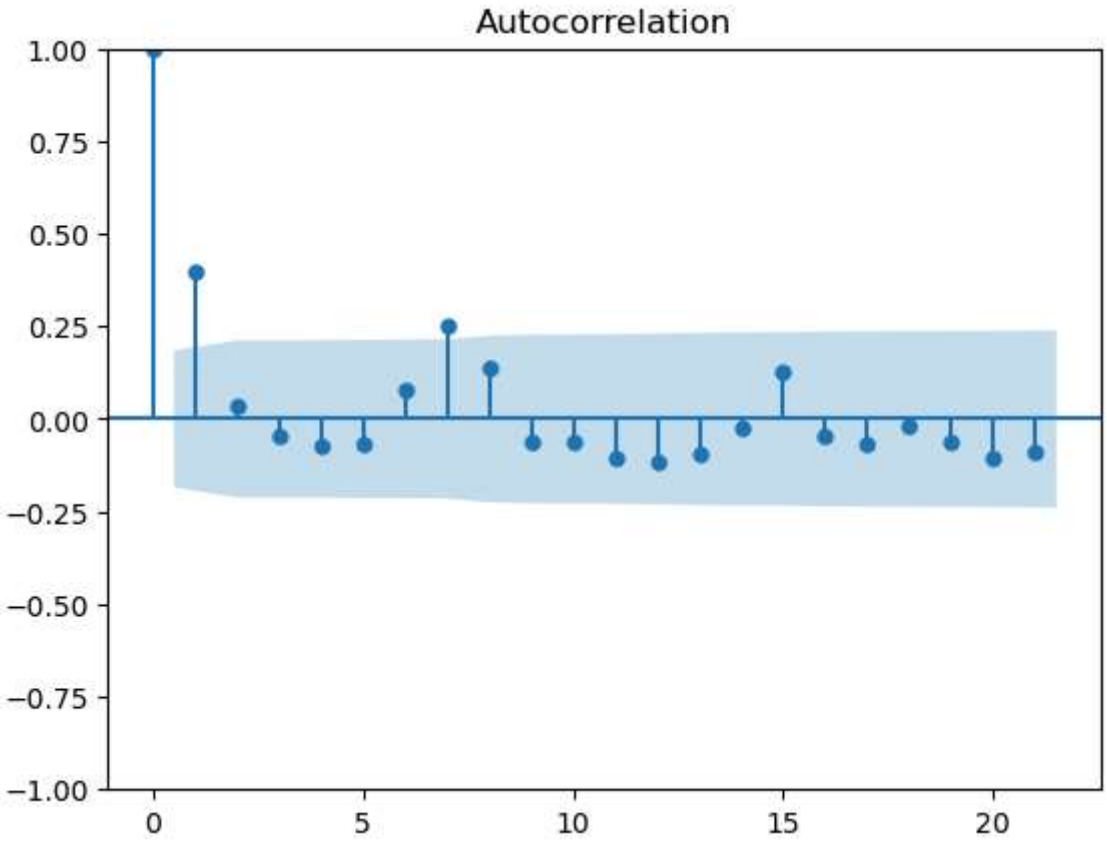
In [38]:
```
# Creating the B ACF plots
acfB = plot_acf(trainB31_endo)

# Creating the B PACF plots
pacfB = plot_pacf(trainB31_endo)
```

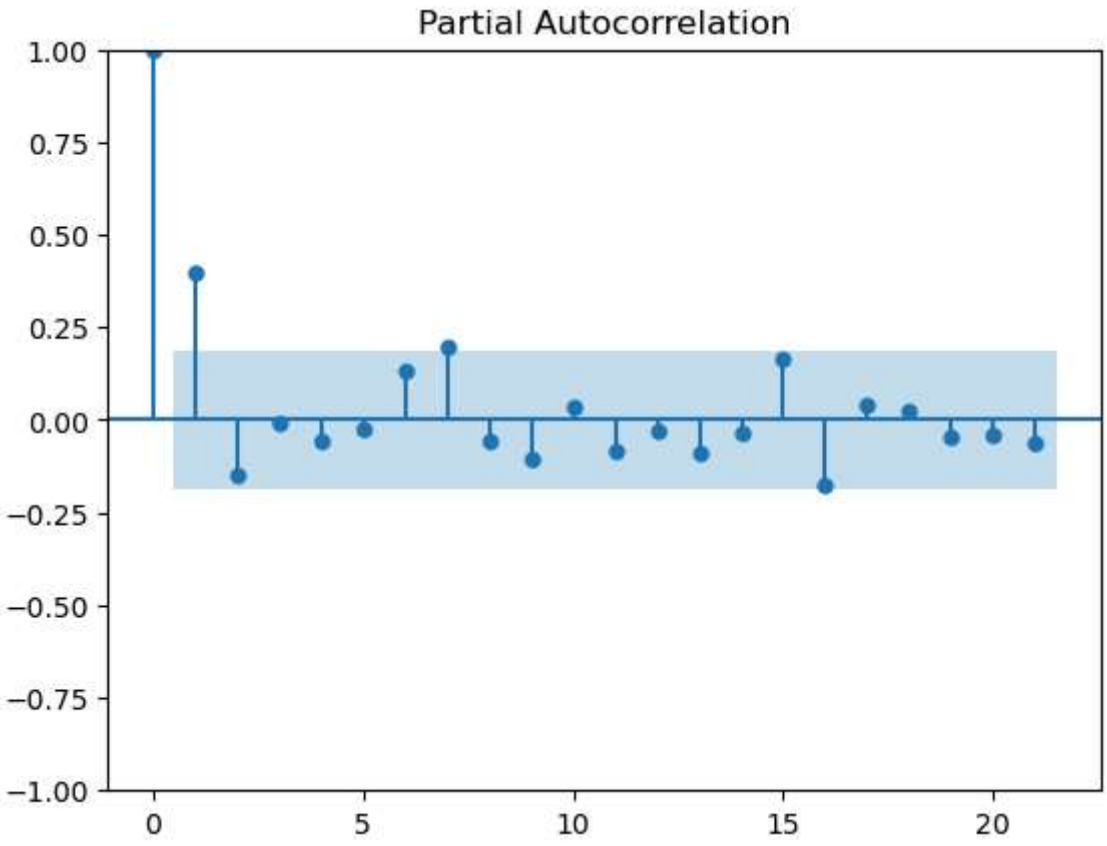




In [39]:
```
# Creating the C ACF plots
acfC = plot_acf(trainC301_endo)
```

```
# Creating the C PACF plots
pacfC = plot_pacf(trainC301_endo)
```





The ACF and PACF charts have significant values at 1 before beginning to converge to 0. The ACF works with the q value and the PACF works with the p value, so q = 1 and p = 1. The nature of the data having multiple stores and departments would make it such that applying differentials to the data would cause some differences to appear that would be from different departments and different stores. So we will set d to be 0.

statsmodels SARIMAX

With the values of p, d, and q found, we can start building our SARIMA model. It would also be wise to standardize the exogenous variables before using them. The values of each field are different from each other and doing so could end up minimizing possible errors.

In [137…

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
import numpy as np

# Making the A model, fitting it, and making predictions
sarimaxA11 = SARIMAX(  # Model
    endog=trainA11_endo,
    exog=trainA11_exo,
    order=(1,0,1),
    seasonal_order=(1,0,1,52))

A11fit = sarimaxA11.fit()  # Fitting
A11preds = A11fit.predict(start='2012-04-06', end='2012-10-26',
                          exog=testA11_exo)  # Predictions

# Making the B model, fitting it, and making predictions
sarimaxB31 = SARIMAX(  # Model
    endog=trainB31_endo,
    exog=trainB31_exo,
    order=(1,0,1),
    seasonal_order=(1,0,1,52))

B31fit = sarimaxB31.fit()  # Fitting
B31preds = B31fit.predict(start='2012-04-06', end='2012-10-26',
                          exog=testB31_exo)  # Predictions

# Making the C model, fitting it, and making predictions
sarimaxC301 = SARIMAX(  # Model
    endog=trainC301_endo,
    exog=trainC301_exo,
    order=(1,0,1),
    seasonal_order=(1,0,1,52))

C301fit = sarimaxC301.fit()  # Fitting
C301preds = C301fit.predict(start='2012-04-06', end='2012-10-26',
                            exog=testC301_exo)  # Predictions
```

```
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency W-FRI will
be used.
  self._init_dates(dates, freq)
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency W-FRI will
be used.
  self._init_dates(dates, freq)
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\statespace\sarimax.py:86
6: UserWarning: Too few observations to estimate starting parameters for seasonal AR
MA. All parameters except for variances will be set to zeros.
  warn('Too few observations to estimate starting parameters%s.'
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency W-FRI will
be used.
  self._init_dates(dates, freq)
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency W-FRI will
be used.
  self._init_dates(dates, freq)
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\statespace\sarimax.py:86
6: UserWarning: Too few observations to estimate starting parameters for seasonal AR
MA. All parameters except for variances will be set to zeros.
  warn('Too few observations to estimate starting parameters%s.'
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\base\model.py:607: Convergenc
eWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency W-FRI will
be used.
  self._init_dates(dates, freq)
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency W-FRI will
be used.
  self._init_dates(dates, freq)
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\tsa\statespace\sarimax.py:86
6: UserWarning: Too few observations to estimate starting parameters for seasonal AR
MA. All parameters except for variances will be set to zeros.
  warn('Too few observations to estimate starting parameters%s.'
C:\Users\dman1\anaconda3\Lib\site-packages\statsmodels\base\model.py:607: Convergenc
eWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

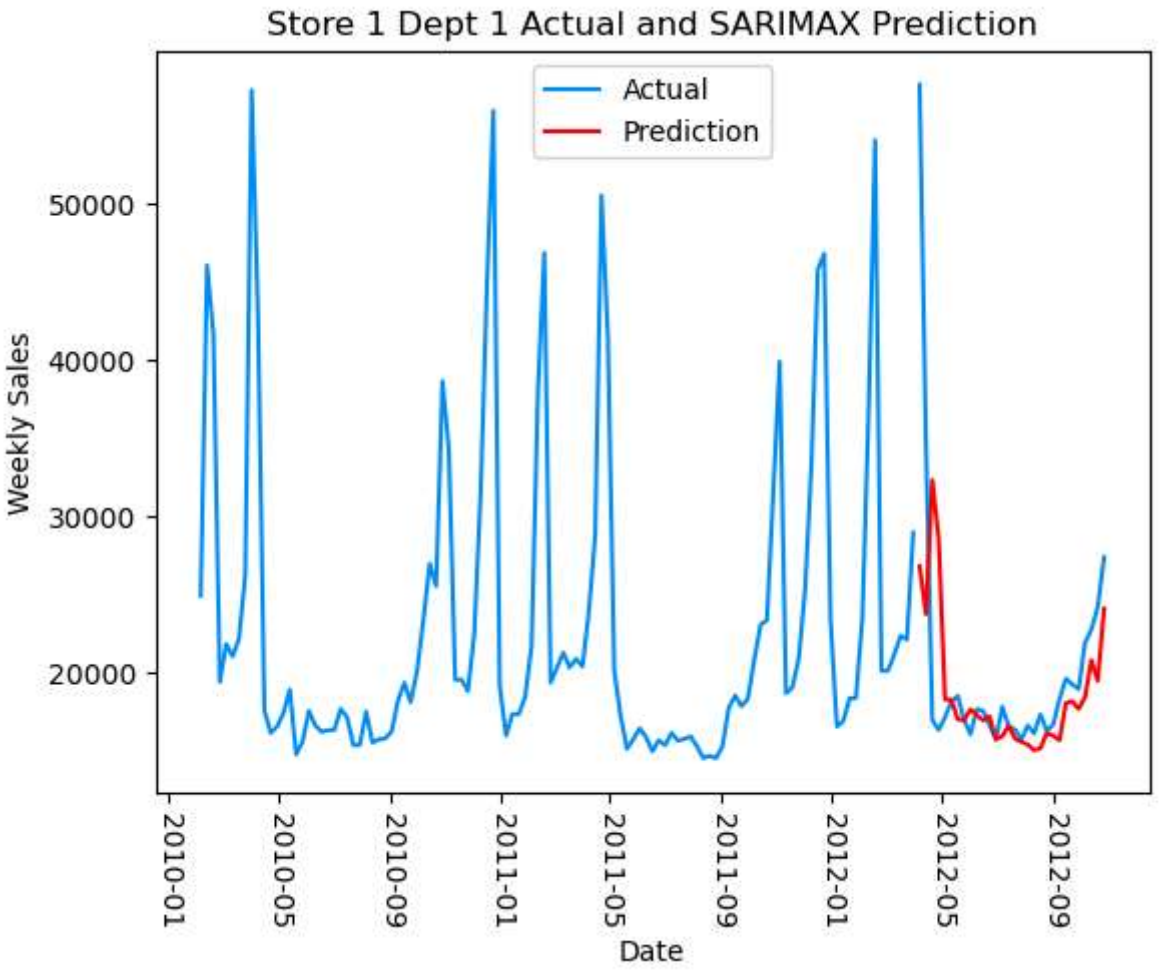
With the models and predictions made, we now will want to evaluate them. We can plot the actual data and the predictions for a graphical representation as well as use statistical metrics like RMSE, MAE, and R-squared.

In [139…

```
plt.plot(testA11_endo, color='#0090F9')
plt.plot(trainA11_endo, color='#0090F9', label='Actual')
plt.plot(A11preds, color='red', label='Prediction')
plt.title('Store 1 Dept 1 Actual and SARIMAX Prediction')
plt.xticks(rotation=-90)
plt.legend()
```

```
plt.xlabel('Date')
plt.ylabel('Weekly Sales')
```

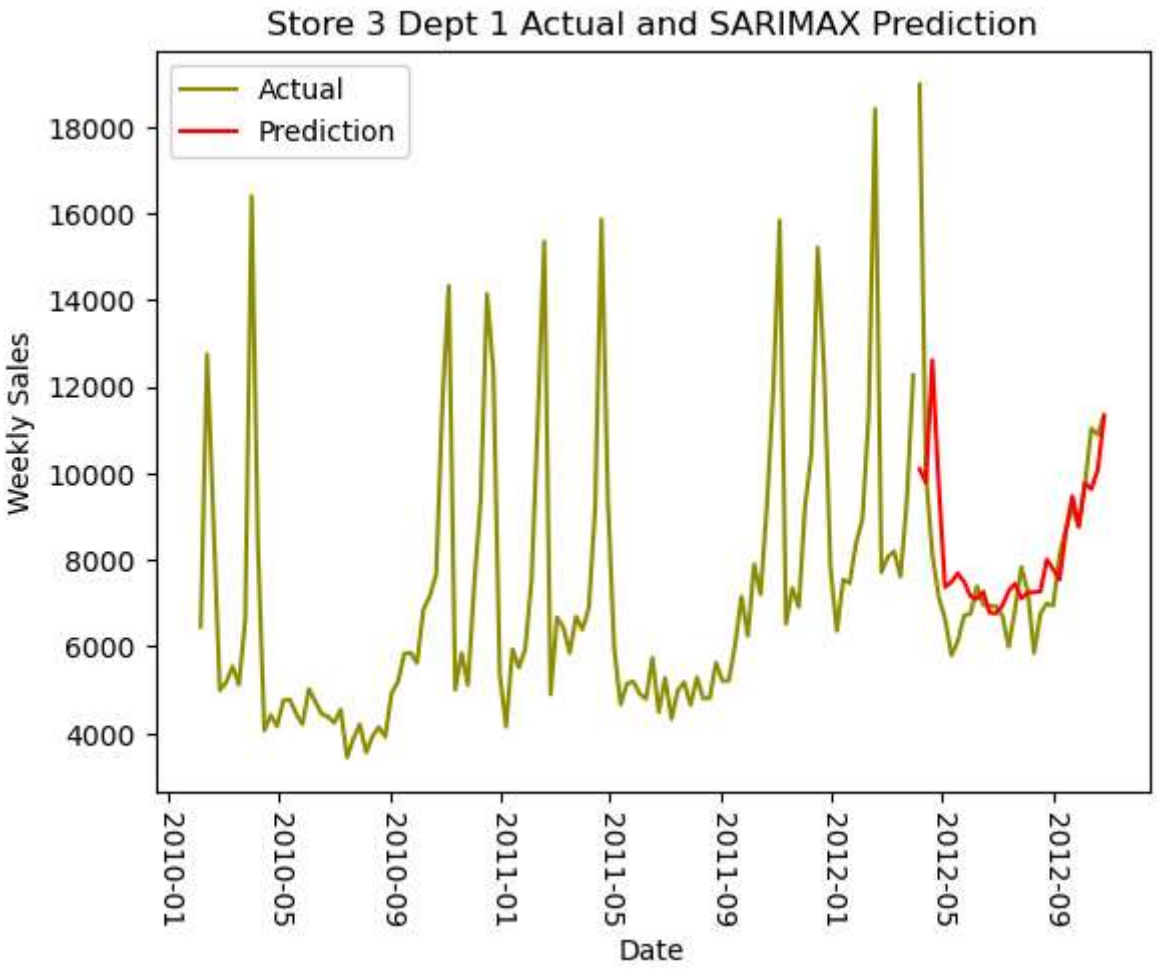
Out[139… Text(0, 0.5, 'Weekly Sales')



In [141…

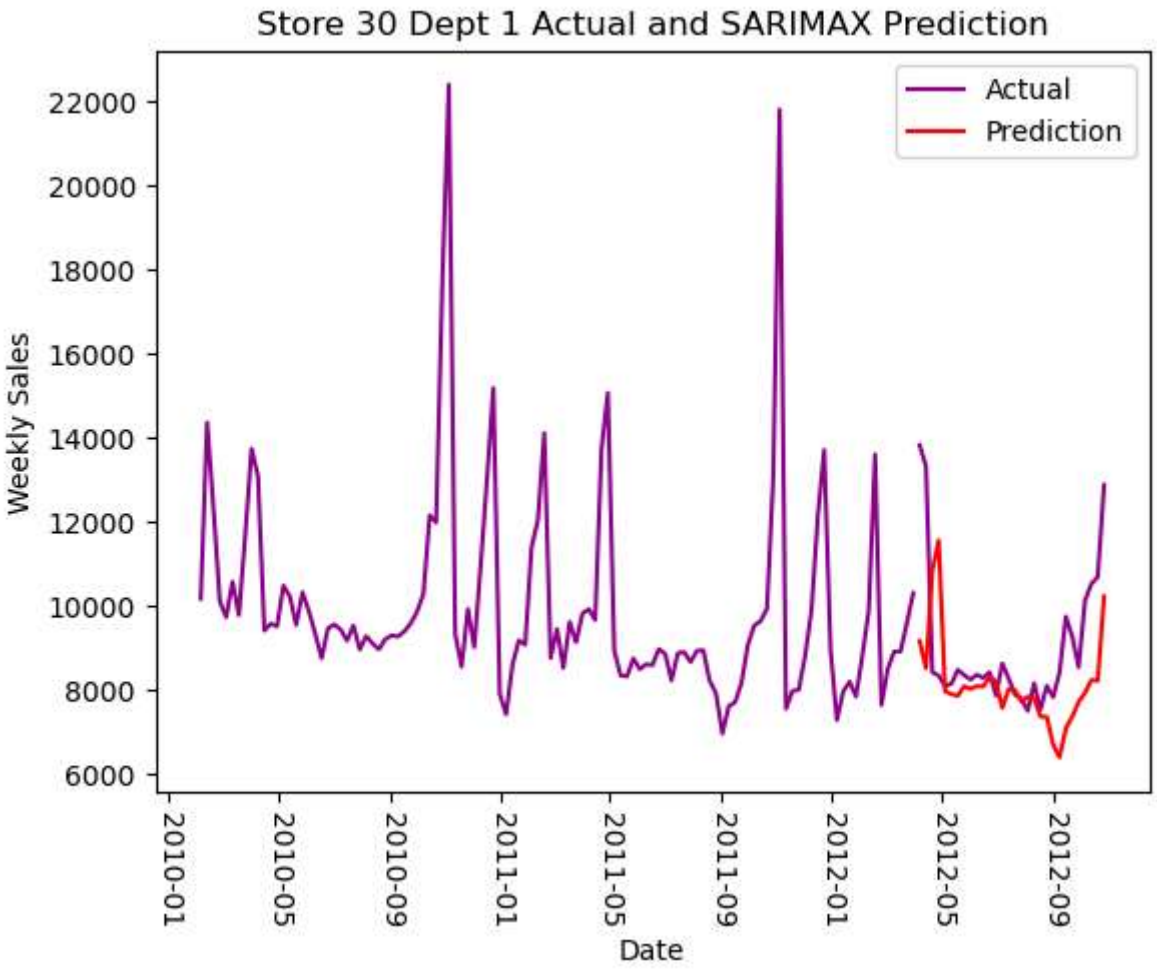
```
plt.plot(testB31_endo, color='#8F9000')
plt.plot(trainB31_endo, color='#8F9000', label='Actual')
plt.plot(B31preds, color='red', label='Prediction')
plt.title('Store 3 Dept 1 Actual and SARIMAX Prediction')
plt.xticks(rotation=-90)
plt.legend()
plt.xlabel('Date')
plt.ylabel('Weekly Sales')
```

Out[141… Text(0, 0.5, 'Weekly Sales')



In [143…
```
plt.plot(testC301_endo, color='#8F008F')
plt.plot(trainC301_endo, color='#8F008F', label='Actual')
plt.plot(C301preds, color='red', label='Prediction' )
plt.title('Store 30 Dept 1 Actual and SARIMAX Prediction')
plt.xticks(rotation=-90)
plt.legend()
plt.xlabel('Date')
plt.ylabel('Weekly Sales')
```

Out[143…
```
Text(0, 0.5, 'Weekly Sales')
```



In [145…

```
from sklearn.metrics import root_mean_squared_error as rmse
from sklearn.metrics import mean_absolute_error as mae
from sklearn.metrics import r2_score as r2

# Collecting Evaluation Stats
A11_rmse = rmse(testA11_endo, A11preds)
A11_mae = mae(testA11_endo, A11preds)
A11r2 = r2(testA11_endo, A11preds)

B31_rmse = rmse(testB31_endo, B31preds)
B31_mae = mae(testB31_endo, B31preds)
B31r2 = r2(testB31_endo, B31preds)

C301_rmse = rmse(testC301_endo, C301preds)
C301_mae = mae(testC301_endo, C301preds)
C301r2 = r2(testC301_endo, C301preds)
```

In [147…

```
# Making Standard deviations for each store's sales
A11stdev = pd.concat([trainA11_endo, testA11_endo]).std()
B31stdev = pd.concat([trainB31_endo, testB31_endo]).std()
C301stdev = pd.concat([trainC301_endo, testC301_endo]).std()

# Getting median values for each store's sales
A11med = pd.concat([trainA11_endo, testA11_endo]).median()
```

```
B31med = pd.concat([trainB31_endo, testB31_endo]).median()
C301med = pd.concat([trainC301_endo, testC301_endo]).median()
```

In [149…

```
# Creating a Metrics Matrix
sarimax_metrics = pd.DataFrame({'Standard Deviation' : [A11stdev, B31stdev,
                                                        C301stdev],
                                'Median' : [A11med, B31med, C301med],
                                'RMSE' : [A11_rmse, B31_rmse, C301_rmse],
                                'MAE' : [A11_mae, B31_mae, C301_mae],
                                'R-Squared' : [A11r2, B31r2, C301r2]})
sarimax_metrics.index=['A11', 'B31', 'C301']
display(sarimax_metrics)
```

	Standard Deviation	Median	RMSE	MAE	R-Squared
A11	9854.349032	18535.48	7145.984275	3437.995561	0.204212
B31	3116.986263	6673.83	2016.296572	1066.618702	0.364688
C301	2373.532654	9164.01	1858.639657	1290.632496	-0.274703

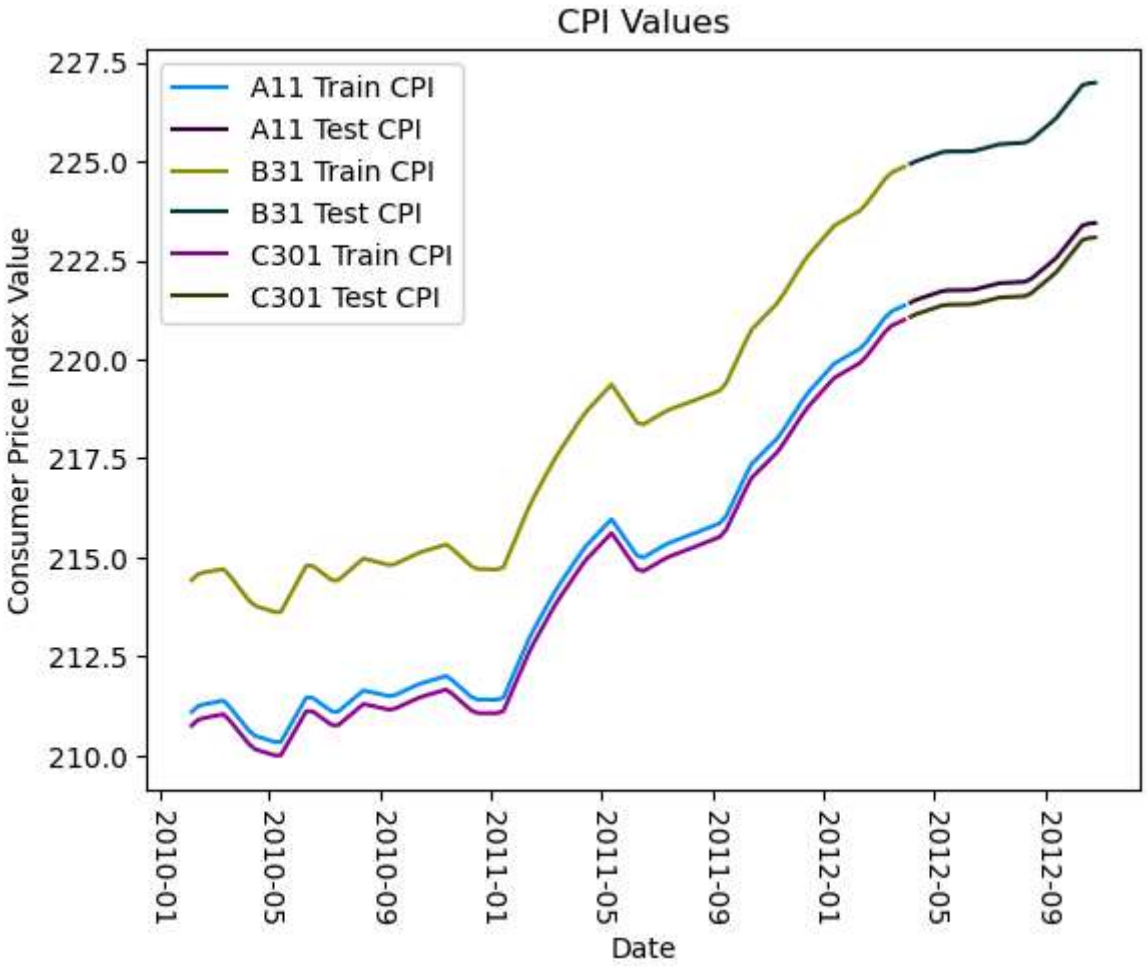
The initial look at the graphs would suggest that the models do follow the trends but aren't doing so very well. The graphs are tending to undershoot the actual retail sales. This could be due to the influence of the exogenous variables on the models themselves. If the exogenous variables after September 2012 were worse but sales did better, it could cause the model to predict lower.

In [121…

```
plt.plot(trainA11_exo['CPI'], color='#0090F9', label='A11 Train CPI')
plt.plot(testA11_exo['CPI'], color ='#360038' , label='A11 Test CPI')

plt.plot(trainB31_exo['CPI'], color='#8F9000', label='B31 Train CPI')
plt.plot(testB31_exo['CPI'], color ='#003838' , label='B31 Test CPI')

plt.plot(trainC301_exo['CPI'], color='#8F008F', label='C301 Train CPI')
plt.plot(testC301_exo['CPI'], color ='#383800' , label='C301 Test CPI')

plt.xticks(rotation=-90)
plt.legend()
plt.ylabel('Consumer Price Index Value')
plt.title('CPI Values')
plt.xlabel('Date')
```

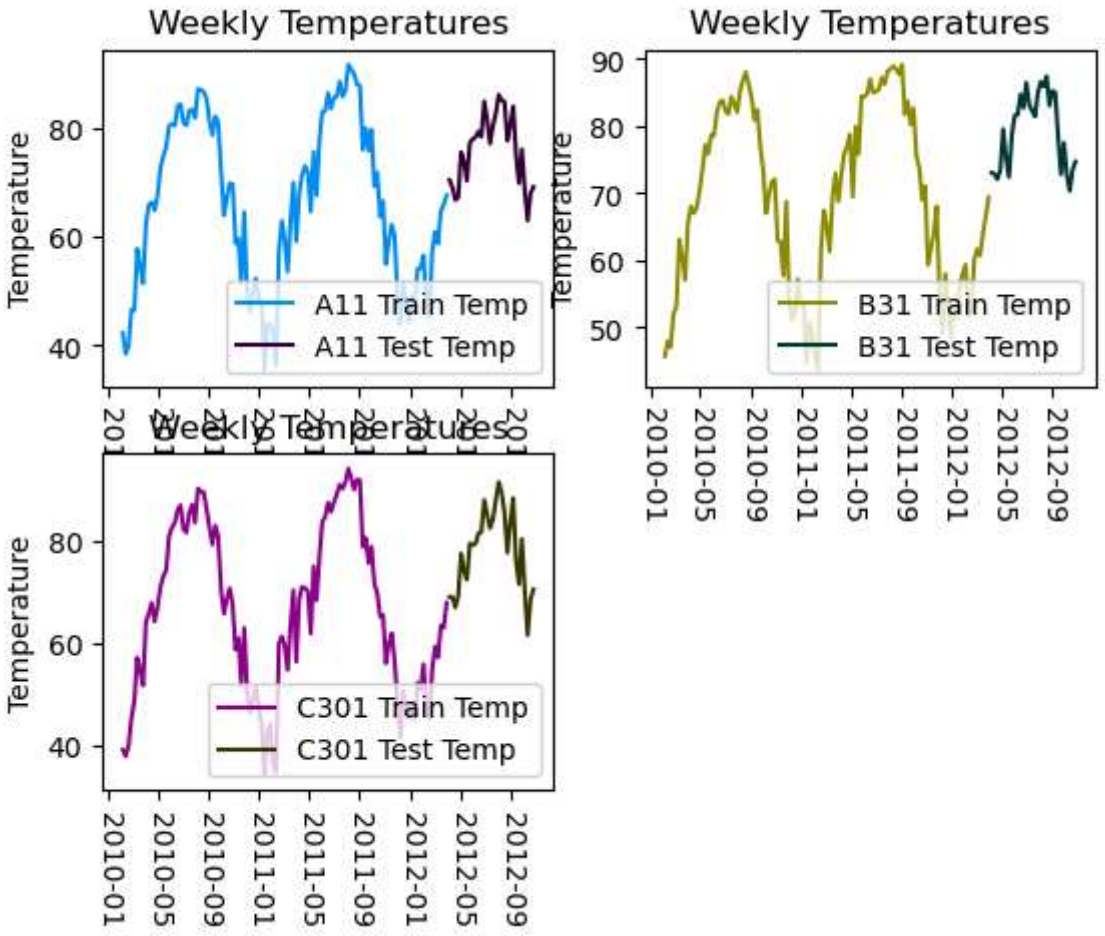
Out[121…

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



In [135…
```
plt.subplot(2,2,1)
plt.plot(trainA11_exo['Temperature'], color='#0090F9', label='A11 Train Temp')
plt.plot(testA11_exo['Temperature'], color ='#360038' , label='A11 Test Temp')
plt.xticks(rotation=-90)
plt.legend()
plt.ylabel('Temperature')
plt.title('Weekly Temperatures')

plt.subplot(2,2,2)
plt.plot(trainB31_exo['Temperature'], color='#8F9000', label='B31 Train Temp')
plt.plot(testB31_exo['Temperature'], color ='#003838' , label='B31 Test Temp')
plt.xticks(rotation=-90)
plt.legend()
plt.ylabel('Temperature')
plt.title('Weekly Temperatures')

plt.subplot(2,2,3)
plt.plot(trainC301_exo['Temperature'], color='#8F008F', label='C301 Train Temp')
plt.plot(testC301_exo['Temperature'], color ='#383800', label='C301 Test Temp')
plt.xticks(rotation=-90)
plt.legend()
plt.ylabel('Temperature')
plt.title('Weekly Temperatures')
```

Out[135…
```
Text(0.5, 1.0, 'Weekly Temperatures')
```

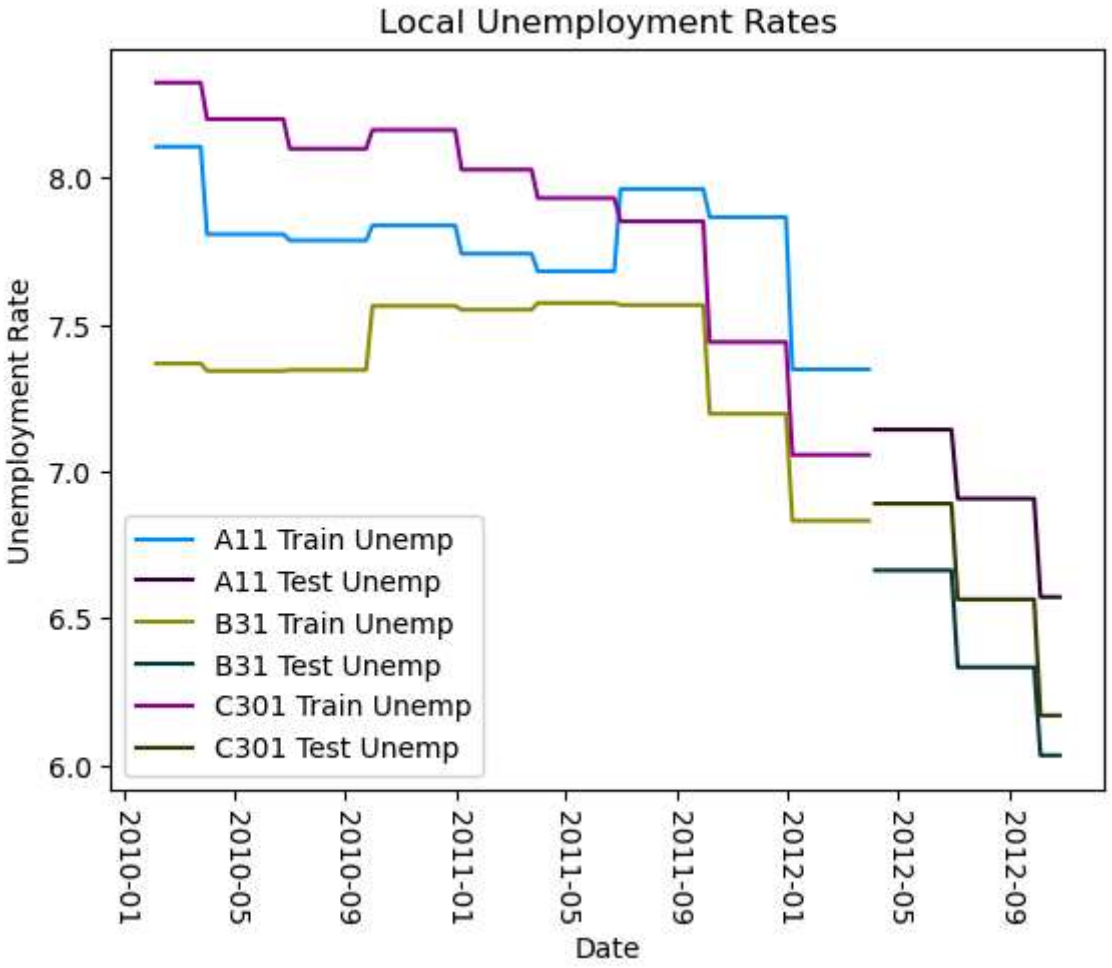


In [123…

```
plt.plot(trainA11_exo['Unemployment'], color='#0090F9', label='A11 Train Unemp')
plt.plot(testA11_exo['Unemployment'], color ='#360038' , label='A11 Test Unemp')

plt.plot(trainB31_exo['Unemployment'], color='#8F9000', label='B31 Train Unemp')
plt.plot(testB31_exo['Unemployment'], color ='#003838' , label='B31 Test Unemp')

plt.plot(trainC301_exo['Unemployment'], color='#8F008F', label='C301 Train Unemp')
plt.plot(testC301_exo['Unemployment'], color ='#383800' , label='C301 Test Unemp')

plt.xticks(rotation=-90)
plt.legend()
plt.ylabel('Unemployment Rate')
plt.title('Local Unemployment Rates')
plt.xlabel('Date')
```

Out[123…

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



In [53]:
```
print(sum(trainA11_exo.Fuel_Price == trainB31_exo.Fuel_Price),
      'A/B Fuel Matches',
      sum(trainA11_exo.Fuel_Price == trainC301_exo.Fuel_Price),
      'A/C Fuel Matches')
```

```
113 A/B Fuel Matches 113 A/C Fuel Matches
```

CPI and temperature values follow similar trends in each for each store. Fuel costs are the same in all of the exogenous variable sets, likely the national average for the week instead of a local average. Local unemployment trends follow similar negative trends after September 2011. It may not be the case that the exogenous variables are what is causing the SARIMAX models to be off. Then again, the fuel costs possibly being a national average instead of a local average could throw off the data, somewhat. It could also be possible that the choices for p, d, and q could be further refined than the semi-arbitrary choices of 1, 0, and 1. This is where pmdarima's SARIMA modeling can come in.

pmdarima SARIMA

pmdarima has an auto-parameterizing ARIMA model that could be used for comparison. The

auto-ARIMA function from pmdarima is different since it has a seasonal parameter in the function instead of using a separate function.

In [56]:
```
import pmdarima as pm

# Creating the auto-SARIMA models
auto_sarimaA11 = pm.auto_arima(
    y= trainA11_endo,
    X= trainA11_exo,
    start_p = 1,
    start_q = 1,
    seasonal = True,
    start_P = 1,
    start_Q = 1,
    m = 52)

auto_sarimaB31 = pm.auto_arima(
    y= trainB31_endo,
    X= trainB31_exo,
    start_p = 1,
    start_q = 1,
    seasonal = True,
    start_P = 1,
    start_Q = 1,
    m = 52)

auto_sarimaC301 = pm.auto_arima(
    y= trainC301_endo,
    X= trainC301_exo,
    start_p = 1,
    start_q = 1,
    seasonal = True,
    start_P = 1,
    start_Q = 1,
    m = 52)
```

In [57]:
```
# Getting auto-sarima model parameters
A11auto_params = auto_sarimaA11.get_params()
B31auto_params = auto_sarimaB31.get_params()
C301auto_params = auto_sarimaC301.get_params()

print('A11 Auto-SARIMA Parameters:', A11auto_params,
     '\n B31 Auto-SARIMA Parameters:', B31auto_params,
     '\n C301 Auto-SARIMA Parameters:', C301auto_params)
```

```
A11 Auto-SARIMA Parameters: {'maxiter': 50, 'method': 'lbfgs', 'order': (0, 0, 3),
'out_of_sample_size': 0, 'scoring': 'mse', 'scoring_args': {}, 'seasonal_order': (1,
0, 0, 52), 'start_params': None, 'suppress_warnings': True, 'trend': None, 'with_int
ercept': False}
 B31 Auto-SARIMA Parameters: {'maxiter': 50, 'method': 'lbfgs', 'order': (2, 0, 1),
'out_of_sample_size': 0, 'scoring': 'mse', 'scoring_args': {}, 'seasonal_order': (1,
0, 0, 52), 'start_params': None, 'suppress_warnings': True, 'trend': None, 'with_int
ercept': False}
 C301 Auto-SARIMA Parameters: {'maxiter': 50, 'method': 'lbfgs', 'order': (0, 0, 1),
'out_of_sample_size': 0, 'scoring': 'mse', 'scoring_args': {}, 'seasonal_order': (1,
0, 0, 52), 'start_params': None, 'suppress_warnings': True, 'trend': None, 'with_int
ercept': False}
```

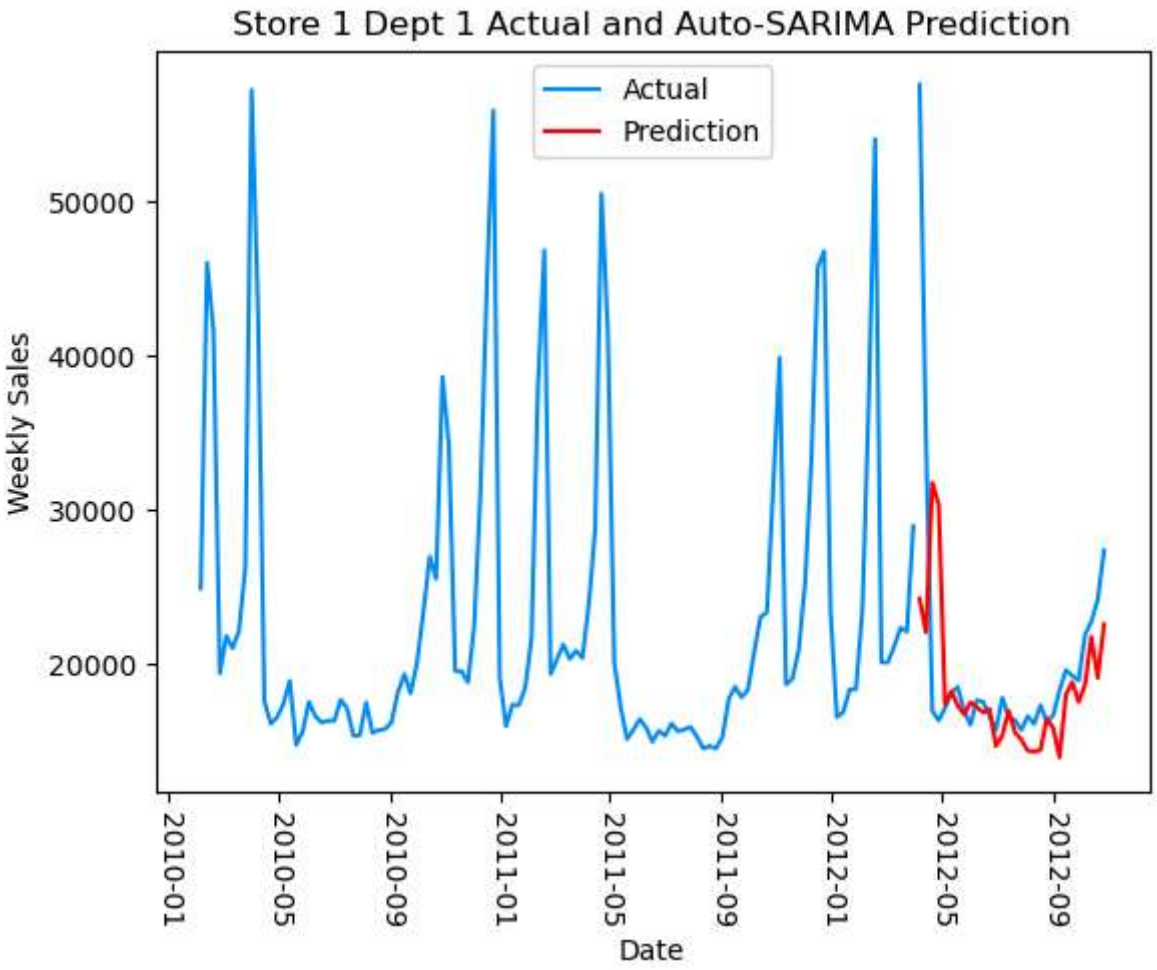
The auto-SARIMA models ended up creating different models with different ARIMA parameters for p and q. The seasonal parameters were the same for each model's P and Q. With the models created, we can make predictions for scoring with RMSE, MAE, and R-squared.

In [59]:
```
# Getting the pmdarima predictions
A11_auto_preds = auto_sarimaA11.predict(y=testA11_endo,
                                        X=testA11_exo,
                                       n_periods=30)
B31_auto_preds = auto_sarimaB31.predict(y=testB31_endo,
                                        X=testB31_exo,
                                       n_periods=30)
C301_auto_preds = auto_sarimaC301.predict(y=testC301_endo,
                                        X=testC301_exo,
                                       n_periods=30)
```

In [60]:
```
# Collecting auto-sarima evaluation stats
autoA11_rmse = rmse(testA11_endo, A11_auto_preds)
autoA11_mae = mae(testA11_endo, A11_auto_preds)
autoA11r2 = r2(testA11_endo, A11_auto_preds)

autoB31_rmse = rmse(testB31_endo, B31_auto_preds)
autoB31_mae = mae(testB31_endo, B31_auto_preds)
autoB31r2 = r2(testB31_endo, B31_auto_preds)

autoC301_rmse = rmse(testC301_endo, C301_auto_preds)
autoC301_mae = mae(testC301_endo, C301_auto_preds)
autoC301r2 = r2(testC301_endo, C301_auto_preds)
```

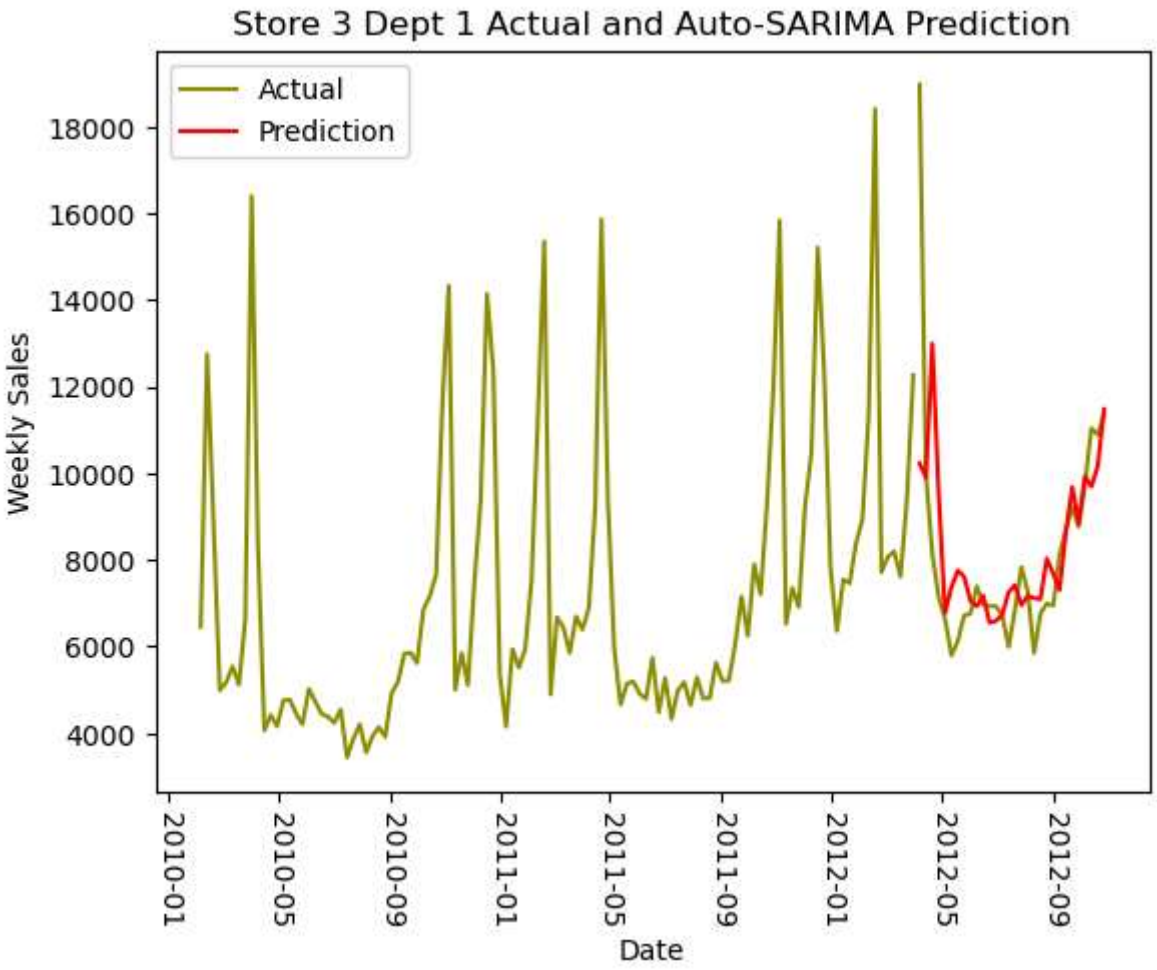
In [119…
```
plt.plot(testA11_endo, color='#0090F9')
plt.plot(trainA11_endo, color='#0090F9', label='Actual')
plt.plot(A11_auto_preds, color='red', label='Prediction')
plt.title('Store 1 Dept 1 Actual and Auto-SARIMA Prediction')
plt.xticks(rotation=-90)
plt.legend()
plt.xlabel('Date')
plt.ylabel('Weekly Sales')
```

Out[119…
```
Text(0, 0.5, 'Weekly Sales')
```



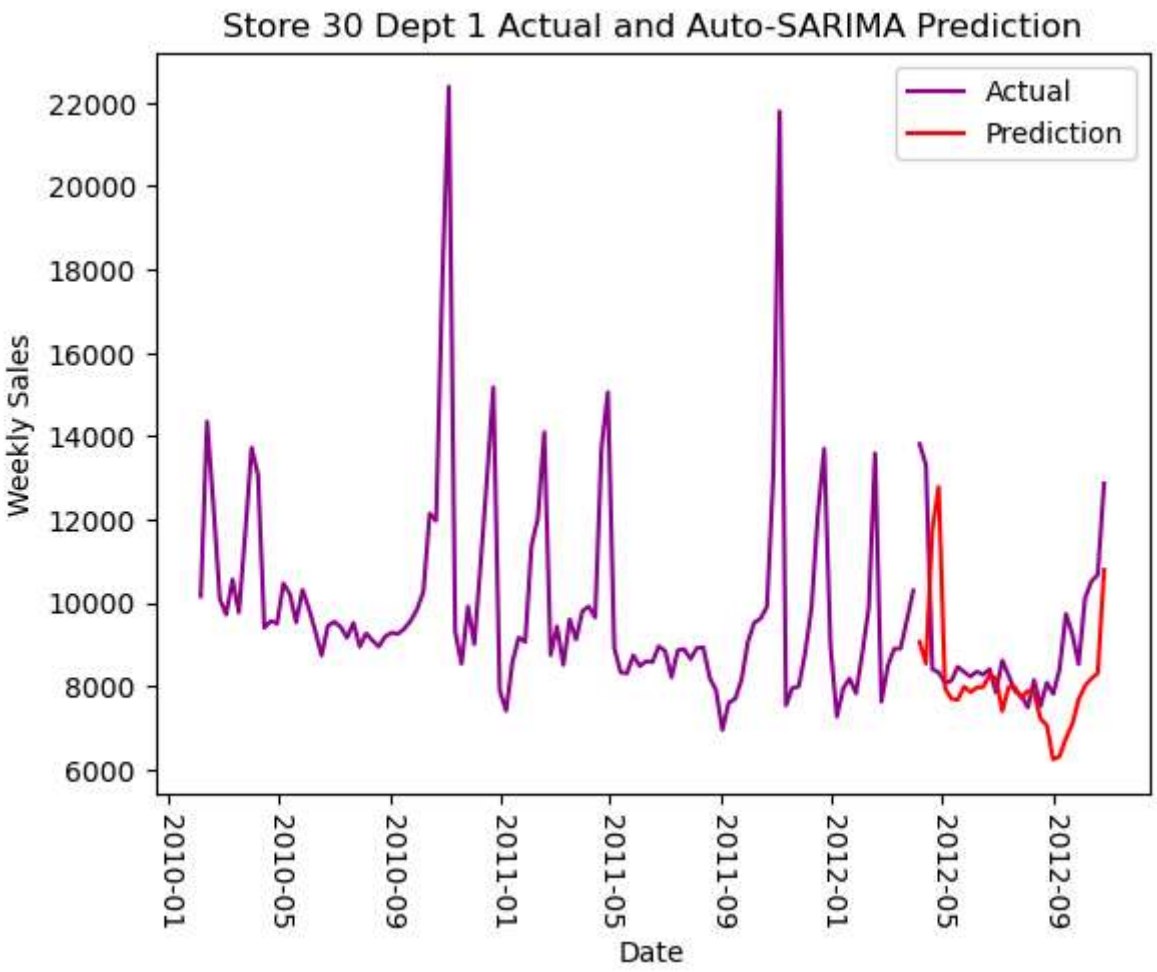
In [117…
```
plt.plot(testB31_endo, color='#8F9000')
plt.plot(trainB31_endo, color='#8F9000', label='Actual')
plt.plot(B31_auto_preds, color='red', label='Prediction')
plt.title('Store 3 Dept 1 Actual and Auto-SARIMA Prediction')
plt.xticks(rotation=-90)
plt.legend()
plt.xlabel('Date')
plt.ylabel('Weekly Sales')
```

Out[117…
```
Text(0, 0.5, 'Weekly Sales')
```



In [115…
```
plt.plot(testC301_endo, color='#8F008F')
plt.plot(trainC301_endo, color='#8F008F', label='Actual')
plt.plot(C301_auto_preds, color='red', label='Prediction' )
plt.title('Store 30 Dept 1 Actual and Auto-SARIMA Prediction')
plt.xticks(rotation=-90)
plt.legend()
plt.xlabel('Date')
plt.ylabel('Weekly Sales')
```

Out[115…
```
Text(0, 0.5, 'Weekly Sales')
```



In [151…
```
# Organizing the metrics to display
auto_sarima_metrics = pd.DataFrame({'Standard Deviation' : [A11stdev, B31stdev,
                                                            C301stdev],
                                    'Median' : [A11med, B31med, C301med],
                                    'RMSE' : [autoA11_rmse, autoB31_rmse,
                                              autoC301_rmse],
                                    'MAE' : [autoA11_mae, autoB31_mae,
                                             autoC301_mae],
                                    'R-Squared' : [autoA11r2, autoB31r2,
                                                   autoC301r2]})
auto_sarima_metrics.index=['A11', 'B31', 'C301']
display(auto_sarima_metrics)
```

	Standard Deviation	Median	RMSE	MAE	R-Squared
A11	9854.349032	18535.48	7746.970916	3824.622660	0.064730
B31	3116.986263	6673.83	2013.368280	1066.505557	0.366532
C301	2373.532654	9164.01	2011.990131	1425.019152	-0.493724

In [153…
```
display(sarimax_metrics)
```

	Standard Deviation	Median	RMSE	MAE	R-Squared
A11	9854.349032	18535.48	7145.984275	3437.995561	0.204212
B31	3116.986263	6673.83	2016.296572	1066.618702	0.364688
C301	2373.532654	9164.01	1858.639657	1290.632496	-0.274703

Comparing the performaces of the models, we can see that the auto-SARIMA models had worse RMSEs. Its MAEs are mixed: the auto-SARIMA had better MAEs for stores 3 and 30 but was worse for store 1. The R-squared values for the auto-SARIMA models were also worse than the SARIMAX R-squared values (though all of them are pretty poor in my opinion). When comparing RMSE and MAE to the standard deviation of the weekly sales however, we can clearly see that the MAE scores better and each metric is within one standard deviation of sales from the combined training and testing periods.

In [ ]: