ACasaccioDSC630 - 8.2 Assignment

May 3, 2024

Author: Alysen Casaccio

DSC630 - Predictive Analytics

Assignment 8.2: Time Series Modeling

Due Date: 5-5-24

[37]:
```
#import statements
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
from sklearn.cluster import KMeans
from sklearn.neighbors import LocalOutlierFactor
from sklearn.ensemble import IsolationForest
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import statsmodels.api as sm
import itertools
from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.statespace.sarimax import SARIMAX
from pmdarima import auto_arima
```

[50]:
```
# suppressing future warnings for this assignment
warnings.filterwarnings('ignore')
```

0.0.1 Data Loading and Basic Cleansing

[4]:
```
# loading and reading
file_path = 'C:/Users/alyse/OneDrive/Documents/Bellevue University/DSC 630 -␣
↪Predictive Analytics/Data Sources/us_retail_sales.csv'
sales = pd.read_csv(file_path)
```

[5]:

```
# converting to long format
df_long = pd.melt(sales, id_vars=['YEAR'], value_vars=sales.columns[1:],␣
 ↪var_name='Month', value_name='Sales')
df_long['Date'] = pd.to_datetime(df_long['YEAR'].astype(str) +␣
 ↪df_long['Month'], format='%Y%b')
df_long.drop(['YEAR', 'Month'], axis=1, inplace=True)
df_long.set_index('Date', inplace=True)
df_long.sort_index(inplace=True)
```

[6]:

```
# checking the data
sales.info()
sales.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 13 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   YEAR    30 non-null     int64
 1   JAN     30 non-null     int64
 2   FEB     30 non-null     int64
 3   MAR     30 non-null     int64
 4   APR     30 non-null     int64
 5   MAY     30 non-null     int64
 6   JUN     30 non-null     int64
 7   JUL     29 non-null     float64
 8   AUG     29 non-null     float64
 9   SEP     29 non-null     float64
 10  OCT     29 non-null     float64
 11  NOV     29 non-null     float64
 12  DEC     29 non-null     float64
dtypes: float64(6), int64(7)
memory usage: 3.2 KB
```

[6]:

```
   YEAR     JAN     FEB     MAR     APR     MAY     JUN       JUL       AUG  \
0  1992  146925  147223  146805  148032  149010  149800  150761.0  151067.0
1  1993  157555  156266  154752  158979  160605  160127  162816.0  162506.0
2  1994  167518  169649  172766  173106  172329  174241  174781.0  177295.0
3  1995  182413  179488  181013  181686  183536  186081  185431.0  186806.0
4  1996  189135  192266  194029  194744  196205  196136  196187.0  196218.0

        SEP       OCT       NOV       DEC
0  152588.0  153521.0  153583.0  155614.0
1  163258.0  164685.0  166594.0  168161.0
2  178787.0  180561.0  180703.0  181524.0
3  187366.0  186565.0  189055.0  190774.0
4  198859.0  200509.0  200174.0  201284.0
```

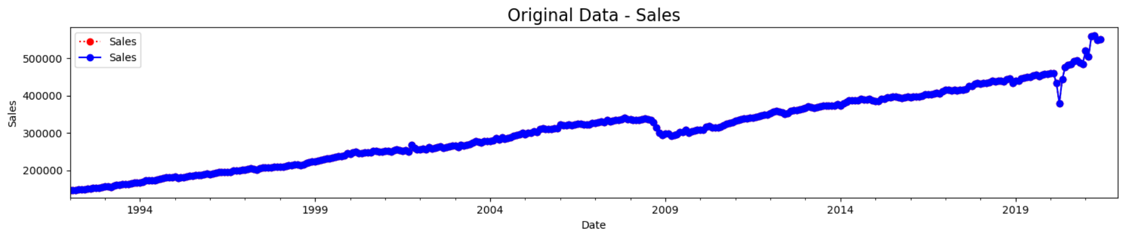
0.0.2 Handling Missing Values

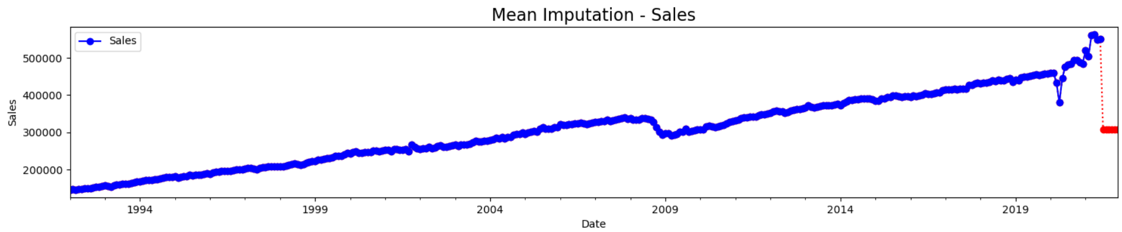
I could see that some of my data was formatted as a float and others were int, but I needed to handle the missing values before converting the datatypes. I was not initially certain of the best method to handle the missing values and chose to construct visualizations to compare how each method would fill the data. Because the choice of missing value input could highly impact my outcome, I wanted to take extra time to assess this choice.

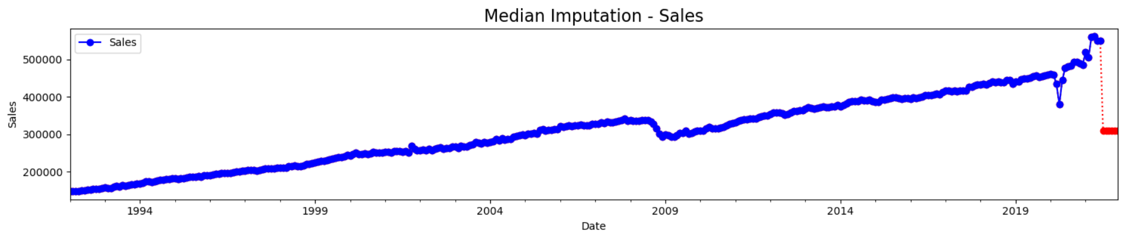
The methods I considered included mean, median, forward fill, backfill, and interpolation using linear, quadratic, and nearest methods.

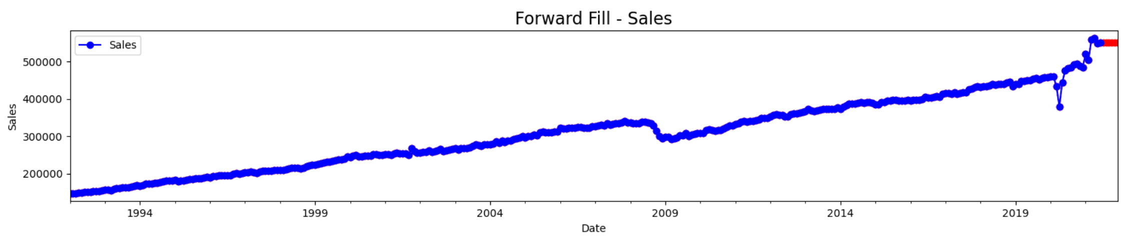
I initially coded and examined these one by one, but streamlined the process here for easy comparison for the submission of my assignment.

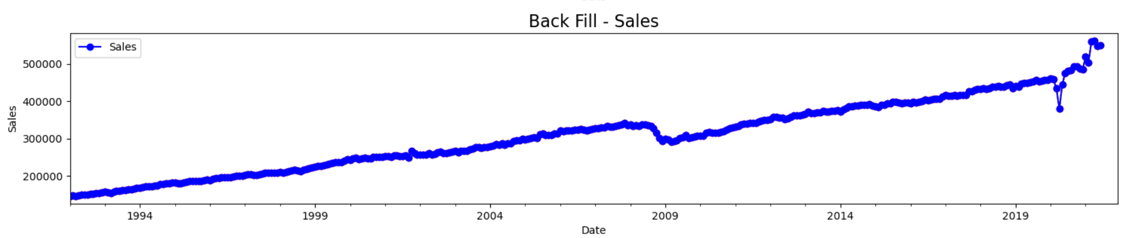
[7]:
```python
# preparing the different methodologies
imputations = {
    'Original Data': df_long,
    'Mean Imputation': df_long['Sales'].fillna(df_long['Sales'].mean()),
    'Median Imputation': df_long['Sales'].fillna(df_long['Sales'].median()),
    'Forward Fill': df_long['Sales'].fillna(method='ffill'),
    'Back Fill': df_long['Sales'].fillna(method='bfill'),
    'Linear Interpolation': df_long['Sales'].interpolate(method='linear'),
    'Quadratic Interpolation': df_long['Sales'].interpolate(method='quadratic'),
    'Nearest Interpolation': df_long['Sales'].interpolate(method='nearest')}

fig, axes = plt.subplots(len(imputations), 1, figsize=(15, 25))

# plotting each method
for ax, (label, series) in zip(axes, imputations.items()):
    series.plot(ax=ax, color='red', marker='o', linestyle='dotted',␣
 ↪title=f"{label} - Sales")
    imputations['Original Data'].plot(ax=ax, marker='o', color='blue')
    ax.set_title(label + ' - Sales', fontsize=16)
    ax.set_ylabel('Sales')

plt.tight_layout()
plt.show()
```

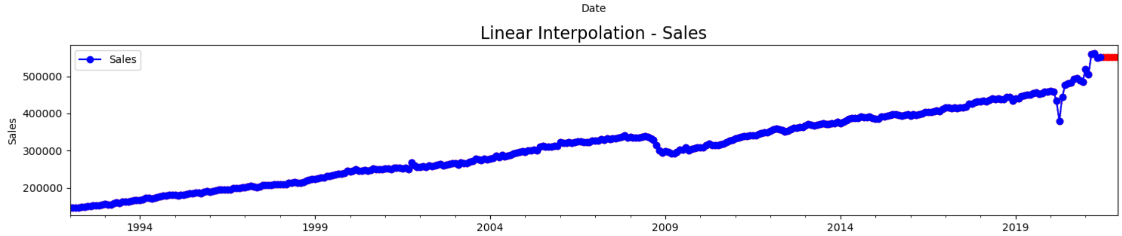

Original Data - Sales
Sales
Sales
500000
400000
300000
200000
Sales
1994
1999
2004
2009
2014
2019
Date

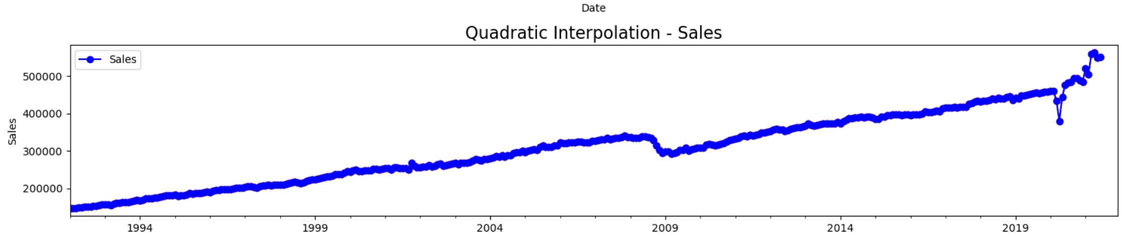

Mean Imputation - Sales
Sales
500000
400000
300000
200000
Sales
1994
1999
2004
2009
2014
2019
Date

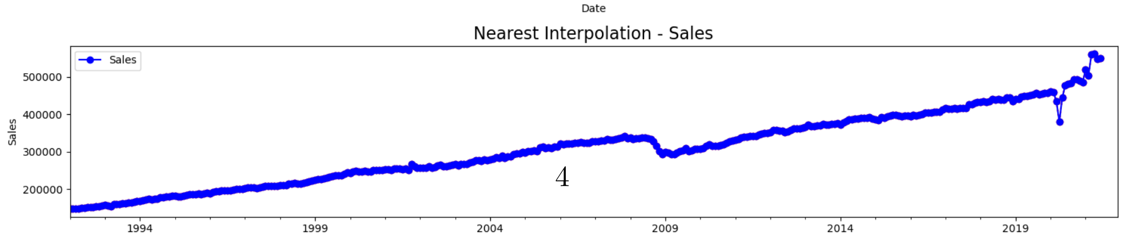

Median Imputation - Sales
Sales
500000
400000
300000
200000
Sales
1994
1999
2004
2009
2014
2019
Date


Forward Fill - Sales
Sales
500000
400000
300000
200000
Sales
1994
1999
2004
2009
2014
2019
Date


Back Fill - Sales
Sales
500000
400000
300000
200000
Sales
1994
1999
2004
2009
2014
2019
Date


Linear Interpolation - Sales
Sales
500000
400000
300000
200000
Sales
1994
1999
2004
2009
2014
2019
Date

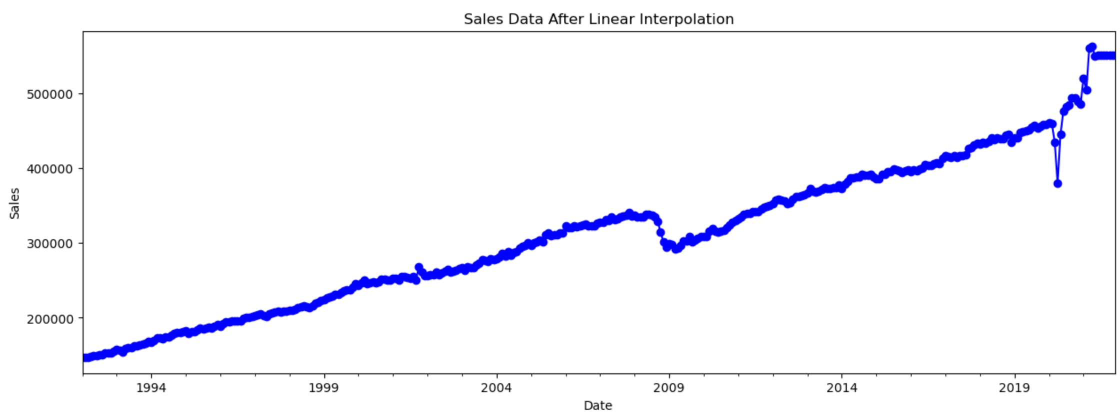

Quadratic Interpolation - Sales
Sales
500000
400000
300000
200000
Sales
1994
1999
2004
2009
2014
2019
Date


Nearest Interpolation - Sales
Sales
500000
400000
300000
200000
Sales
1994
1999
2004
2009
2014
2019
Date

Missing Values Interpretation In my initial attempts at this assignment, I was replacing the missing values with a mean. Reviewing the visualizations on this above clearly indicate that I was creating more outliers with this method and missing the nuance of the trending over time. After appropriate assessment, I decided to move forward with linear interpolation instead.

[8]:
```
# filling missing data using linear interpolation
df_long['Sales'] = df_long['Sales'].interpolate(method='linear')

# Optionally, check if there are any remaining missing values
print("Remaining NaN after interpolation:", df_long['Sales'].isna().sum())

# Plotting the result to visualize how interpolation has filled the NaN values
plt.figure(figsize=(15, 5))
df_long['Sales'].plot(marker='o', linestyle='-', color='blue')
plt.title('Sales Data After Linear Interpolation')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.show()
```

```
Remaining NaN after interpolation: 0
```

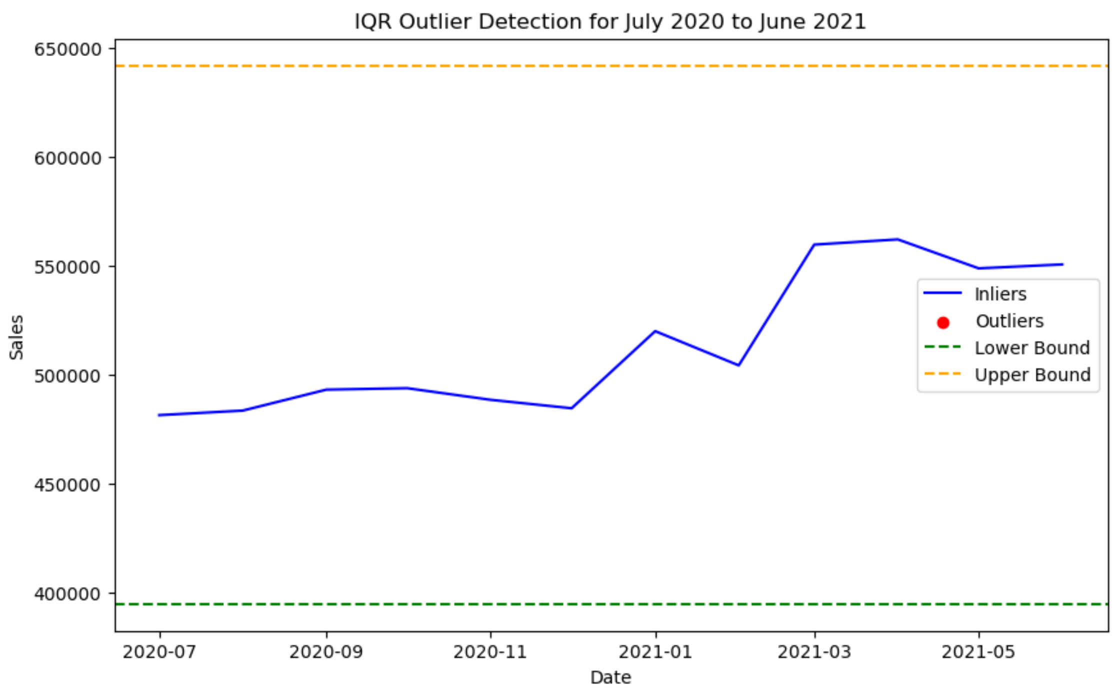


0.0.3 Handling Outliers

Much in the same way, I decided to examine multiple methods of identifying and managing outliers as part of pre-processing before running any predictive models. The methods of outlier detection I chose were interquartile range (IQR), K-means, and local outlier factor (LOF). As these methods were more complex than managing missing data, I left them each in their own coding cell. Initially, I checked for outliers across the entire dataset and decided to leave them in as a conservative approach.

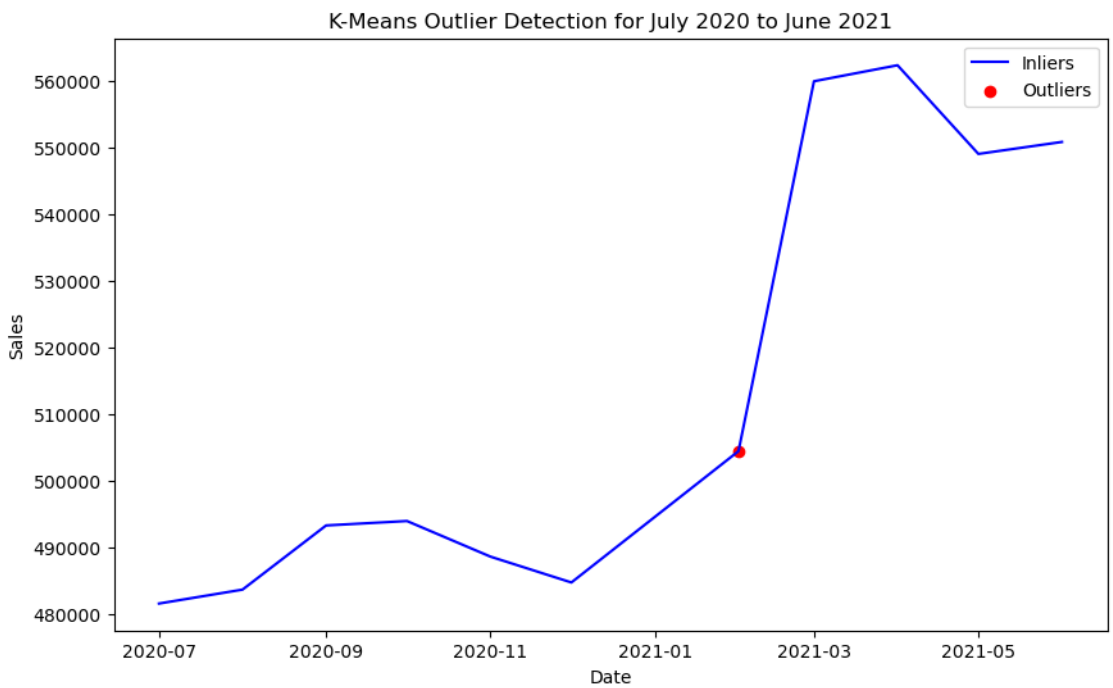
After running a linear regression, a Holt's Linear Trend, and two iterations of SARIMA models, I decided to return to my outlier determinations and focus only on the months of the test data. Those filtered outlier detection attempts are below.

[41]:
```
# filtering the data for the specific period
period_data = df_long['2020-07-01':'2021-06-30']

# computing the IQR-based bounds
Q1 = period_data['Sales'].quantile(0.25)
Q3 = period_data['Sales'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# identifying outlier indices within the specified period
outlier_indices = period_data[(period_data['Sales'] < lower_bound) |␣
↪(period_data['Sales'] > upper_bound)].index

# plotting the data with outliers marked in red
plt.figure(figsize=(10, 6))
plt.plot(period_data.index, period_data['Sales'], color='blue', label='Inliers')
plt.scatter(outlier_indices, period_data.loc[outlier_indices, 'Sales'],␣
↪color='red', label='Outliers')
plt.axhline(y=lower_bound, color='green', linestyle='--', label='Lower Bound')
plt.axhline(y=upper_bound, color='orange', linestyle='--', label='Upper Bound')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('IQR Outlier Detection for July 2020 to June 2021')
plt.legend()
plt.show()
```



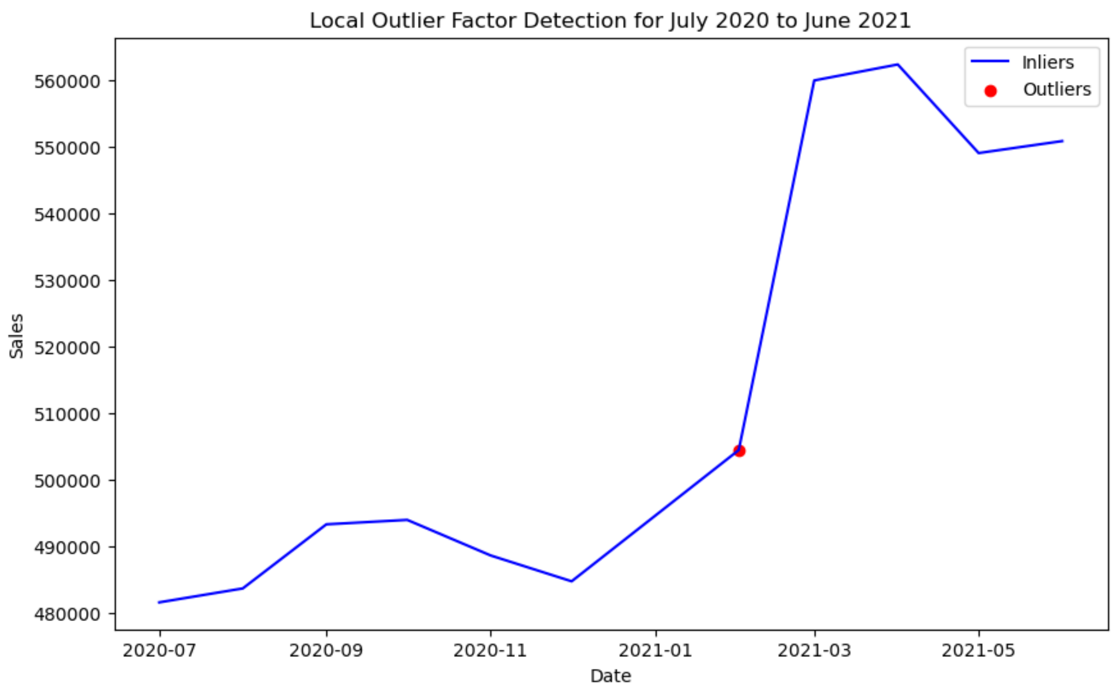
[51]:
```python
# filtering the data for the specific period
period_data = df_long['2020-07-01':'2021-06-30']

# reshaping data for K-Means
X = period_data[['Sales']].values

# creating a KMeans model
n_clusters = 2  # Adjust based on your specific data
model = KMeans(n_clusters=n_clusters, random_state=42)
model.fit(X)

# assigning each data point to a cluster
cluster_assignments = model.predict(X)
cluster_centers = model.cluster_centers_
distances = np.linalg.norm(X - cluster_centers[cluster_assignments], axis=1)

# defining a threshold to identify outliers
threshold = np.percentile(distances, 99)

# identifying outlier indices
outlier_indices = np.where(distances > threshold)[0]

# plotting the data with outliers marked in red
plt.figure(figsize=(10, 6))
```

```
plt.plot(period_data.index, X, color='blue', label='Inliers')
plt.scatter(period_data.index[outlier_indices], X[outlier_indices],␣
  ↪color='red', label='Outliers')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('K-Means Outlier Detection for July 2020 to June 2021')
plt.legend()
plt.show()
```



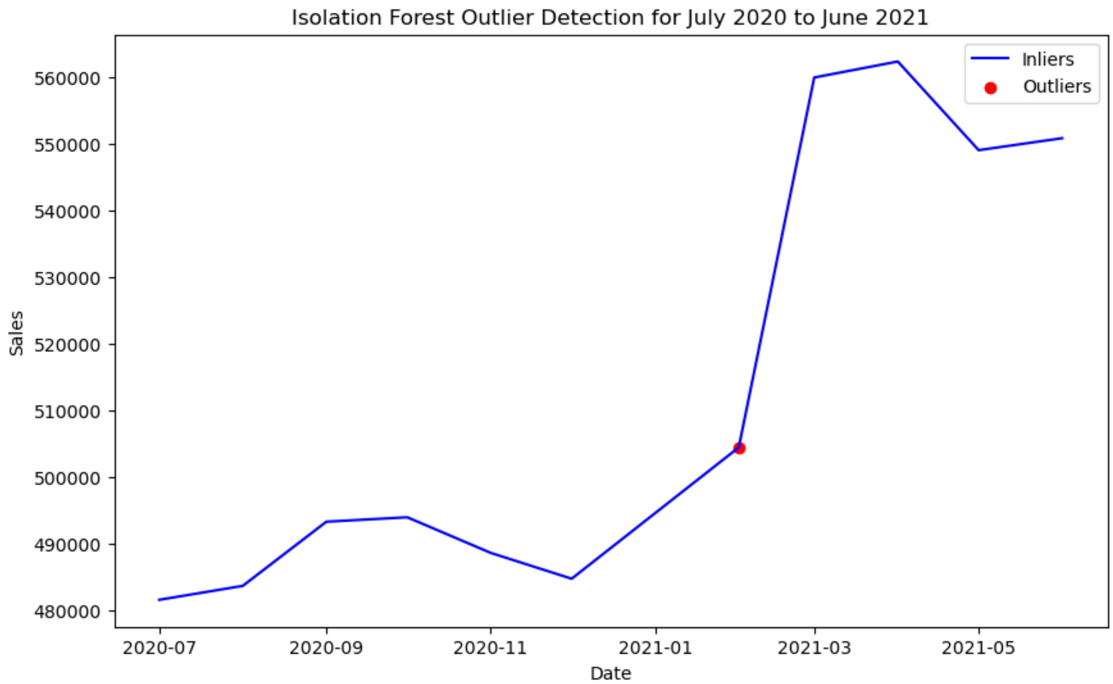
[52]:
```
# filtering the data for the specific period
period_data = df_long['2020-07-01':'2021-06-30']

# reshaping data for LOF
X = period_data[['Sales']].values

# creating a LOF model
model = LocalOutlierFactor(n_neighbors=20, contamination=0.05)
outlier_scores = model.fit_predict(X)

# identifying outlier indices
outliers = period_data[outlier_scores == -1]

# plotting the data and marking outliers in red
plt.figure(figsize=(10, 6))
```

```
plt.plot(period_data.index, period_data['Sales'], color='blue', label='Inliers')
plt.scatter(outliers.index, outliers['Sales'], color='red', label='Outliers')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('Local Outlier Factor Detection for July 2020 to June 2021')
plt.legend()
plt.show()
```



[54]:
```
# filtering the data for the specific period
period_data = df_long['2020-07-01':'2021-06-30']

# creating the model
model = IsolationForest(n_estimators=100, contamination=0.05, random_state=42)
model.fit(period_data[['Sales']])

# detecting outliers
outlier_scores = model.predict(period_data[['Sales']])

# plotting the data and marking outliers in red
plt.figure(figsize=(10, 6))
plt.plot(period_data.index, period_data['Sales'], color='blue', label='Inliers')
plt.scatter(period_data.index[outlier_scores == -1],␣
↪period_data['Sales'][outlier_scores == -1], color='red', label='Outliers')
plt.xlabel('Date')
```

```
plt.ylabel('Sales')
plt.title('Isolation Forest Outlier Detection for July 2020 to June 2021')
plt.legend()
plt.show()
```



[13]:
```
# checking the data
sales.info()
sales.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 13 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   YEAR    30 non-null     int64
 1   JAN     30 non-null     int64
 2   FEB     30 non-null     int64
 3   MAR     30 non-null     int64
 4   APR     30 non-null     int64
 5   MAY     30 non-null     int64
 6   JUN     30 non-null     int64
 7   JUL     29 non-null     float64
 8   AUG     29 non-null     float64
 9   SEP     29 non-null     float64
 10  OCT     29 non-null     float64
```

```
 11   NOV     29 non-null     float64
 12   DEC     29 non-null     float64
dtypes: float64(6), int64(7)
memory usage: 3.2 KB
```

[13]:
```
   YEAR     JAN     FEB     MAR     APR     MAY     JUN       JUL       AUG  \
0  1992  146925  147223  146805  148032  149010  149800  150761.0  151067.0
1  1993  157555  156266  154752  158979  160605  160127  162816.0  162506.0
2  1994  167518  169649  172766  173106  172329  174241  174781.0  177295.0
3  1995  182413  179488  181013  181686  183536  186081  185431.0  186806.0
4  1996  189135  192266  194029  194744  196205  196136  196187.0  196218.0

        SEP       OCT       NOV       DEC
0  152588.0  153521.0  153583.0  155614.0
1  163258.0  164685.0  166594.0  168161.0
2  178787.0  180561.0  180703.0  181524.0
3  187366.0  186565.0  189055.0  190774.0
4  198859.0  200509.0  200174.0  201284.0
```

[14]:
```python
# handling any remaining NaNs
if sales.isnull().any().any():
    sales.interpolate(method='linear', inplace=True)  # applying interpolation␣
 ↪just in case

# converting all columns to int
sales = sales.astype(int)

# checking the changes
print(sales.info())
print(sales.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 13 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   YEAR    30 non-null     int32
 1   JAN     30 non-null     int32
 2   FEB     30 non-null     int32
 3   MAR     30 non-null     int32
 4   APR     30 non-null     int32
 5   MAY     30 non-null     int32
 6   JUN     30 non-null     int32
 7   JUL     30 non-null     int32
 8   AUG     30 non-null     int32
 9   SEP     30 non-null     int32
 10  OCT     30 non-null     int32
 11  NOV     30 non-null     int32
```

```
 12  DEC     30 non-null     int32
dtypes: int32(13)
memory usage: 1.7 KB
None
   YEAR     JAN     FEB     MAR     APR     MAY     JUN     JUL     AUG  \
0  1992  146925  147223  146805  148032  149010  149800  150761  151067
1  1993  157555  156266  154752  158979  160605  160127  162816  162506
2  1994  167518  169649  172766  173106  172329  174241  174781  177295
3  1995  182413  179488  181013  181686  183536  186081  185431  186806
4  1996  189135  192266  194029  194744  196205  196136  196187  196218

      SEP     OCT     NOV     DEC
0  152588  153521  153583  155614
1  163258  164685  166594  168161
2  178787  180561  180703  181524
3  187366  186565  189055  190774
4  198859  200509  200174  201284
```

Pre-Processing for Time Series Models

[15]:
```
# determining time frequency of the data
time_diff = df_long.index.to_series().diff()
most_common_freq = time_diff.mode().iloc[0]
print(f"Data Frequency: {most_common_freq}")
```

```
Data Frequency: 31 days 00:00:00
```

[16]:
```
# resampling and interpolating to prepare data, especially for the models below
df_resampled = df_long.resample('M').mean()  # resampling to the end of each␣
↪month
df_resampled.interpolate(method='linear', inplace=True)  # interpolating␣
↪missing values
```

[17]:
```
# after resampling/interpolating, checking for stationarity
result = adfuller(df_resampled['Sales'])
print('ADF Statistic:', result[0])
print('p-value:', result[1])
```

```
ADF Statistic: 1.0890322196491957
p-value: 0.9951219214614785
```

Pre-processing Interpretation The positive value of the ADF statistic and a high p-value suggest that the time series data is likely non-stationary.
This isn't surprising given the clear upward trend. The ARIMA model requires stationary data and since I didn't want to remove/flatten the trending, I decided to avoid the ARIMA model and try three predictive models in order: linear regression (to obtain baseline), Holt's Linear Trend (just in case there is only an upward trend and no true seasonality), and then a SARIMA with Trend model (if I actually need to account for both seasonality and trending).

[18]:
```
# splitting data into training and testing
train = df_long[df_long.index < '2020-07']
test = df_long[(df_long.index >= '2020-07') & (df_long.index < '2021-07')]
```

0.0.4 Linear Regression

[19]:
```
# prepping data by converting datetime index to numeric for the regression␣
↪analysis
X_train = np.array((train.index - train.index.min()).days).reshape(-1, 1)
y_train = train['Sales']
X_test = np.array((test.index - test.index.min()).days).reshape(-1, 1)
y_test = test['Sales']

# fitting the model
model = LinearRegression()
model.fit(X_train, y_train)
```

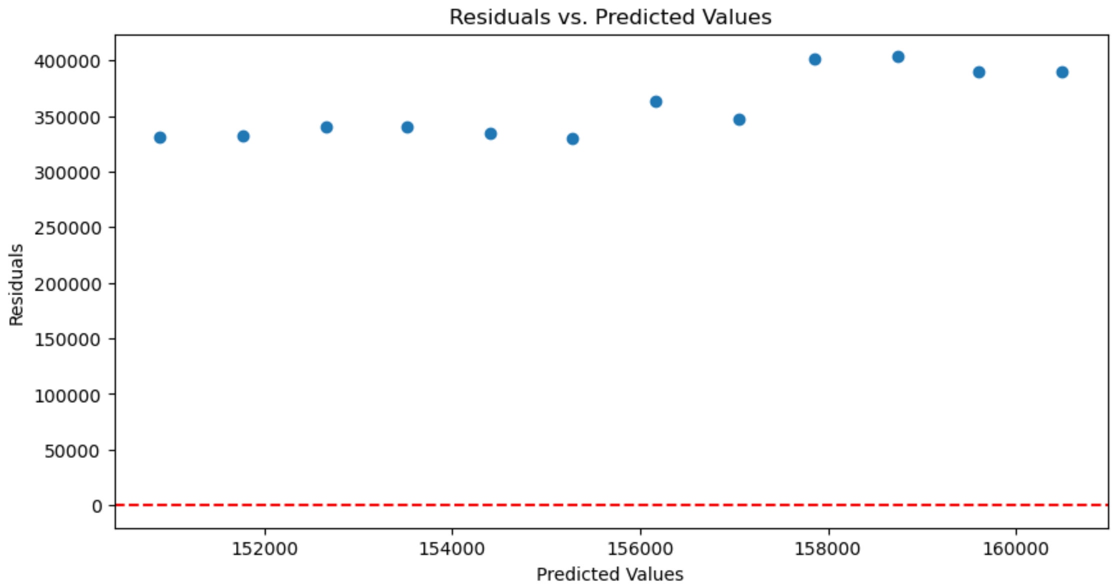
[19]:
```
LinearRegression()
```

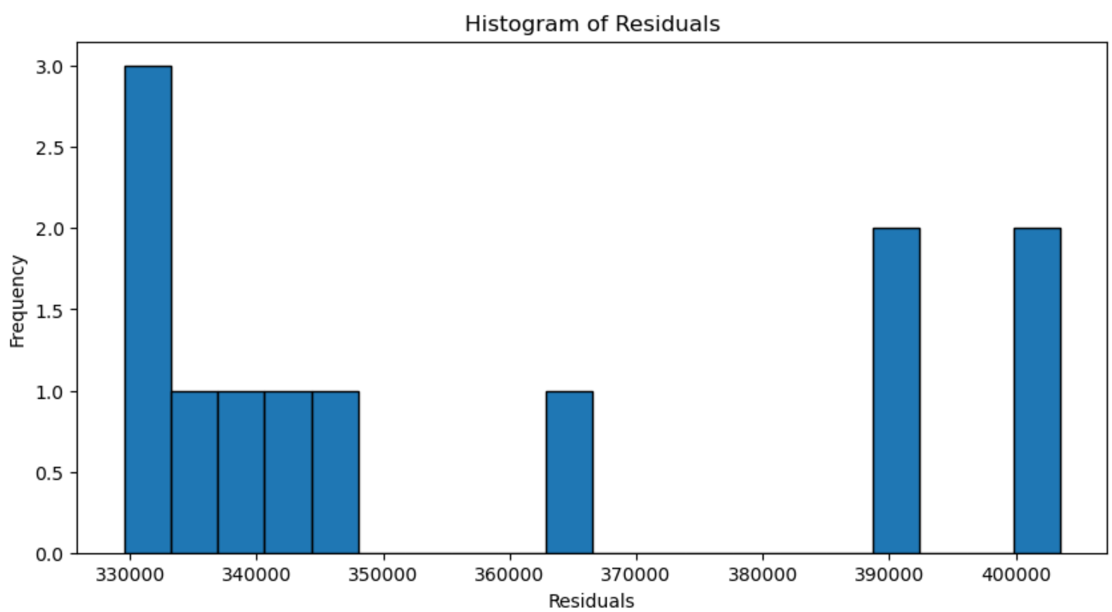
[20]:
```
# predicting
predictions = model.predict(X_test)
```

[21]:
```
# evaluating the linear regression model
rmse = np.sqrt(mean_squared_error(y_test, predictions))
mae = mean_absolute_error(y_test, predictions)
mape = np.mean(np.abs((y_test - predictions) / y_test)) * 100
r_squared = r2_score(y_test, predictions)
print("Root Mean Squared Error (RMSE):", rmse)
print("Mean Absolute Error (MAE):", mae)
print("Mean Absolute Percentage Error (MAPE):", mape, "%")
print("R-squared:", r_squared)

# plotting residuals
residuals = y_test - predictions
plt.figure(figsize=(10, 5))
plt.scatter(predictions, residuals)
plt.title('Residuals vs. Predicted Values')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.axhline(y=0, color='r', linestyle='--')
plt.show()

# histogram of residuals
plt.figure(figsize=(10, 5))
plt.hist(residuals, bins=20, edgecolor='black')
plt.title('Histogram of Residuals')
plt.xlabel('Residuals')
```

```
plt.ylabel('Frequency')
plt.show()
```

```
Root Mean Squared Error (RMSE): 359794.72411175835
Mean Absolute Error (MAE): 358682.8536347481
Mean Absolute Percentage Error (MAPE): 69.6542153018392 %
R-squared: -134.93918975992963
```





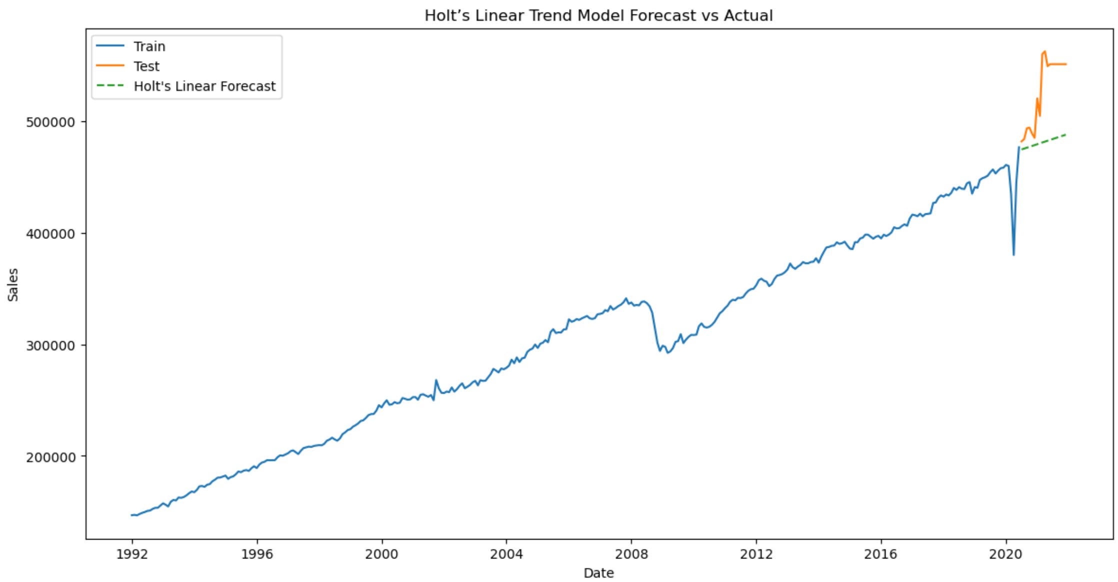
Interpretation of Linear Regression The combination of diagnostics indicated this was not a strong model for the data, and as a result, it had poor predictive performance.
The RMSE and MAE are both so high they are nearly at the scale of the actual sales figures. The MAPE indicates that predictions deviate from actuals by 69.65% on average.
The R-square is so far from a perfect fit, it actually indicates that this model fits worse than a horizontal line representing the mean. I knew the linear regression wouldn't be the best model to use, but I hoped it would have produced an easy-to-interpret baseline from which to improve on. Moving to the next model was the best course of action.

0.0.5 Holt's Linear Trend Model

[22]:
```
# converting data to float for this model
df_long['Sales'] = df_long['Sales'].astype(float)

# setting the index with a monthly start frequency
df_long.index = pd.date_range(start=df_long.index.min(), periods=len(df_long),␣
↪freq='MS')

# splitting the data
train = df_long[df_long.index < '2020-07']
test = df_long[df_long.index >= '2020-07']
```

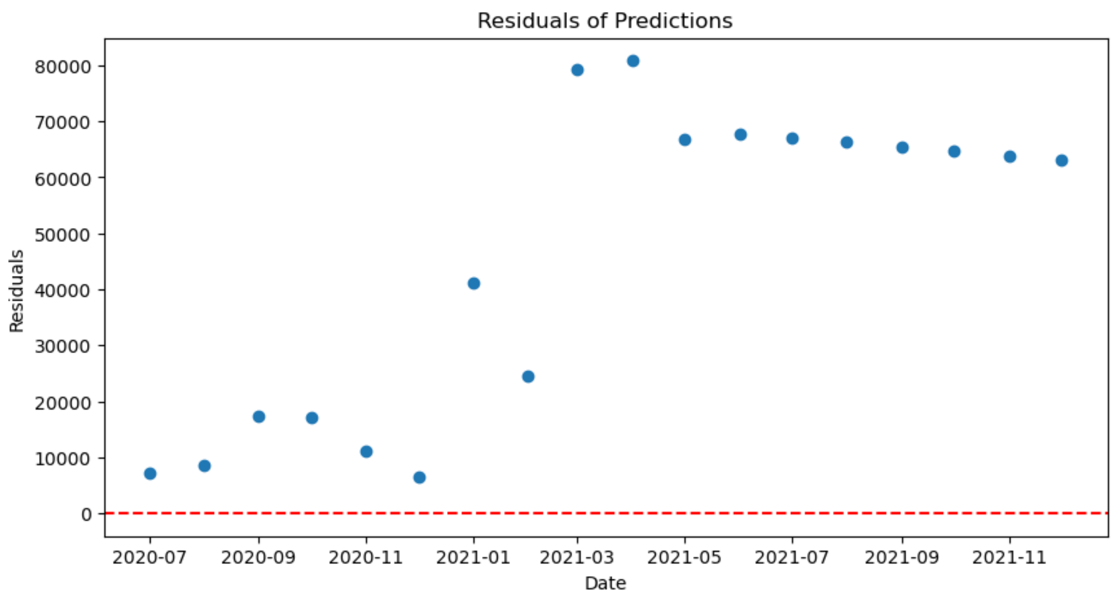
[53]:
```
# fitting Holt's Linear Trend Model
model = ExponentialSmoothing(train['Sales'], trend='add', seasonal=None,␣
↪damped_trend=False, seasonal_periods=None)
fit = model.fit(optimized=True)
```

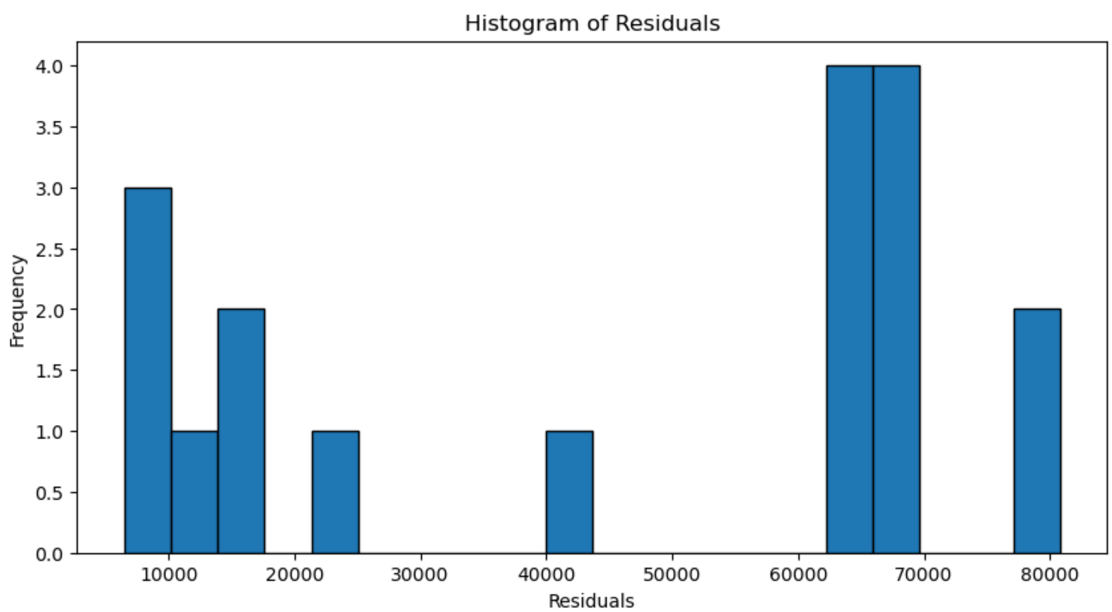
[24]:
```
# forecasting
forecast = fit.forecast(len(test))

# plotting
plt.figure(figsize=(14, 7))
plt.plot(train.index, train['Sales'], label='Train')
plt.plot(test.index, test['Sales'], label='Test')
plt.plot(test.index, forecast, label='Holt\'s Linear Forecast', linestyle='--')
plt.title('Holt's Linear Trend Model Forecast vs Actual')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.show()
```



[25]:
```
# evaluating
rmse = np.sqrt(mean_squared_error(test['Sales'], forecast))
mae = mean_absolute_error(test['Sales'], forecast)
mape = np.mean(np.abs((test['Sales'] - forecast) / test['Sales'])) * 100
print("Root Mean Squared Error (RMSE):", rmse)
print("Mean Absolute Error (MAE):", mae)
print("Mean Absolute Percentage Error (MAPE):", mape, "%")

# calculating residuals
residuals = test['Sales'] - forecast

# plotting residuals
plt.figure(figsize=(10, 5))
plt.scatter(test.index, residuals)
plt.title('Residuals of Predictions')
plt.xlabel('Date')
plt.ylabel('Residuals')
plt.axhline(y=0, color='r', linestyle='--')
plt.show()

# histogram of residuals
plt.figure(figsize=(10, 5))
plt.hist(residuals, bins=20, edgecolor='black')
plt.title('Histogram of Residuals')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.show()
```

```
Root Mean Squared Error (RMSE): 52946.873413229114
Mean Absolute Error (MAE): 45474.20902909077
Mean Absolute Percentage Error (MAPE): 8.361778411333004 %
```





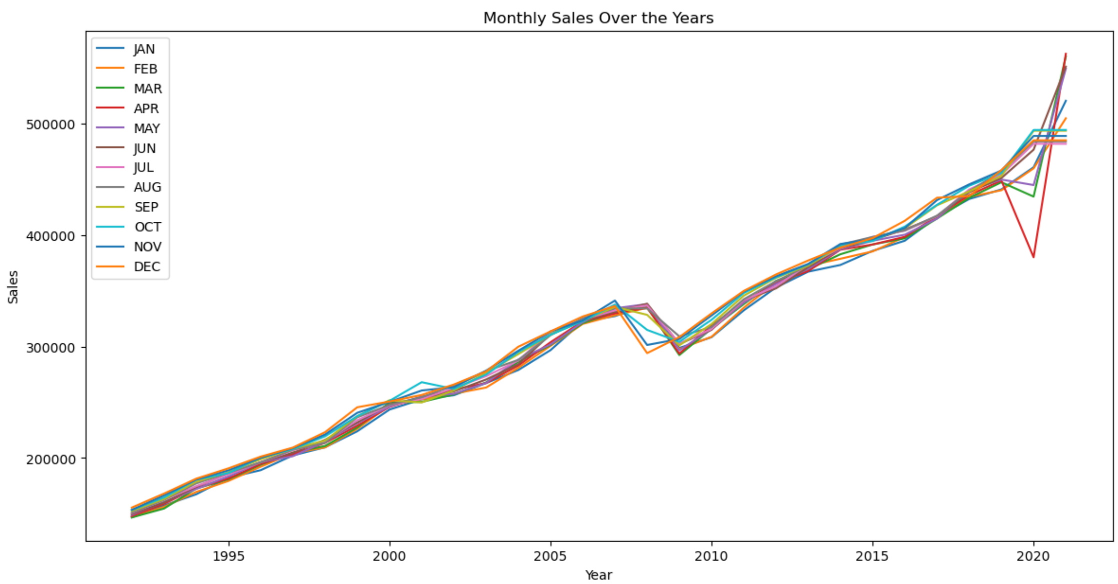
Holt's Linear Trend Interpretation The RMSE was considerably lower than in the linear regression and the MAPE of 8.36% was reassuring. At this point, my model predictions were within 8.36% of the actual sales values. That felt like a strong performance indicator for business

forecasting!
The residuals were still showing potential issues with the model's ability to capture patterns and changes in the trend as time progressed, however.
While I could adjust the damped trend in the Holt's model, to seer if that helped, the apparent limitations led me to move onto a SARIMA model to see if I could get improvement over the Holt.

0.0.6 SARIMA Model – Iteration One

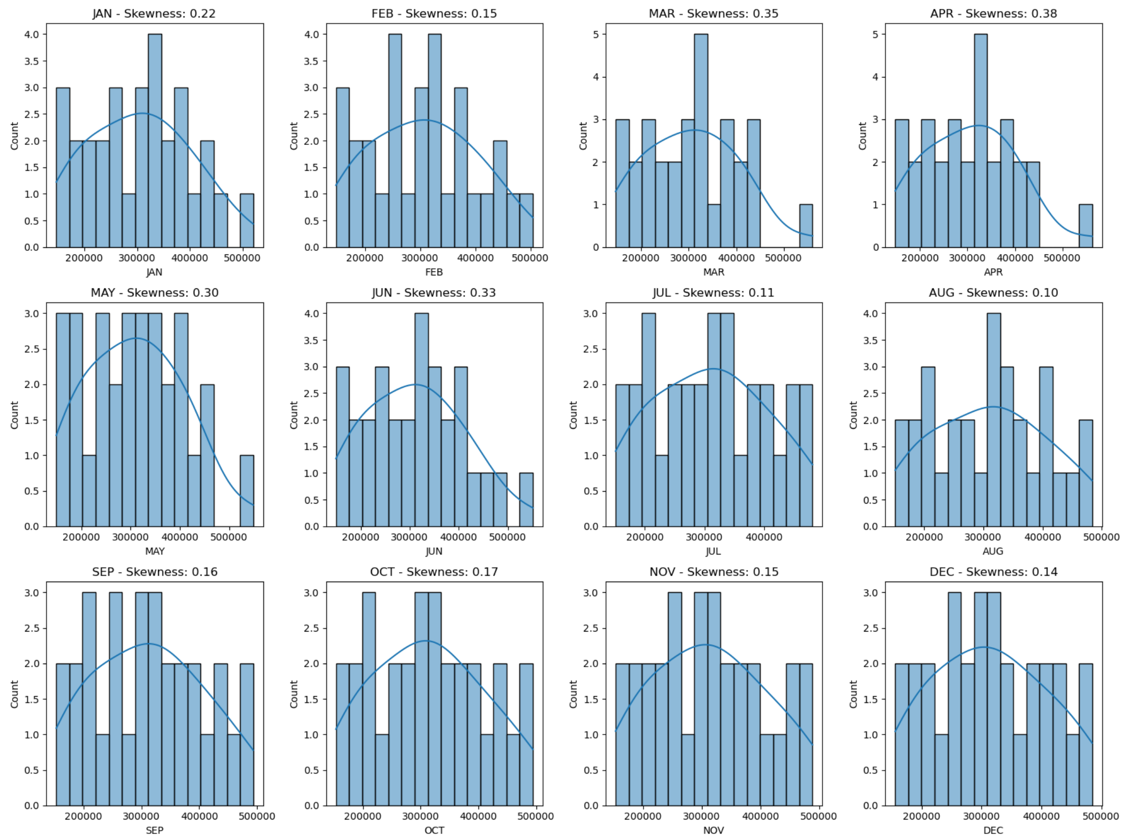
Before executing on the SARIMA model, I wanted to take a moment to look closer at the potential seasonality, distribution, and skewness in the dataset.

[26]:
```
# plotting monthly sales over the years to check for seasonality
plt.figure(figsize=(14, 7))
for month in sales.columns[1:]:
    plt.plot(sales['YEAR'], sales[month], label=month)
plt.title('Monthly Sales Over the Years')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.legend()
plt.show()
```



[27]:
```
# plotting histograms and calculating skewness for each column
fig, axes = plt.subplots(nrows=3, ncols=4, figsize=(16, 12))
axes = axes.flatten()

for i, month in enumerate(sales.columns[1:]):
    sns.histplot(sales[month], bins=15, kde=True, ax=axes[i])
    axes[i].set_title(f'{month} - Skewness: {sales[month].skew():.2f}')
```

```
plt.tight_layout()
plt.show()
```

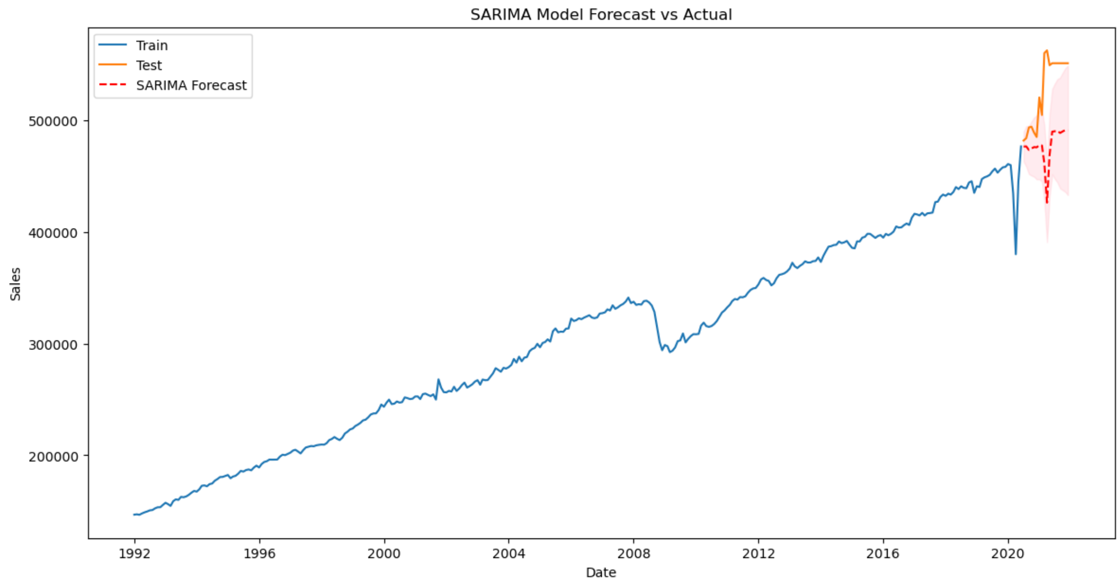


[32]:
```
# fitting the SARIMA
model = SARIMAX(train['Sales'], order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
results = model.fit(disp=False)
```

[33]:
```
# forecasting
forecasts = results.get_forecast(steps=len(test))
forecast_mean = forecasts.predicted_mean
forecast_ci = forecasts.conf_int()
```

[34]:
```
# plotting
plt.figure(figsize=(14, 7))
plt.plot(train.index, train['Sales'], label='Train')
plt.plot(test.index, test['Sales'], label='Test')
plt.plot(test.index, forecast_mean, label='SARIMA Forecast', color='red',␣
 ↪linestyle='--')
```

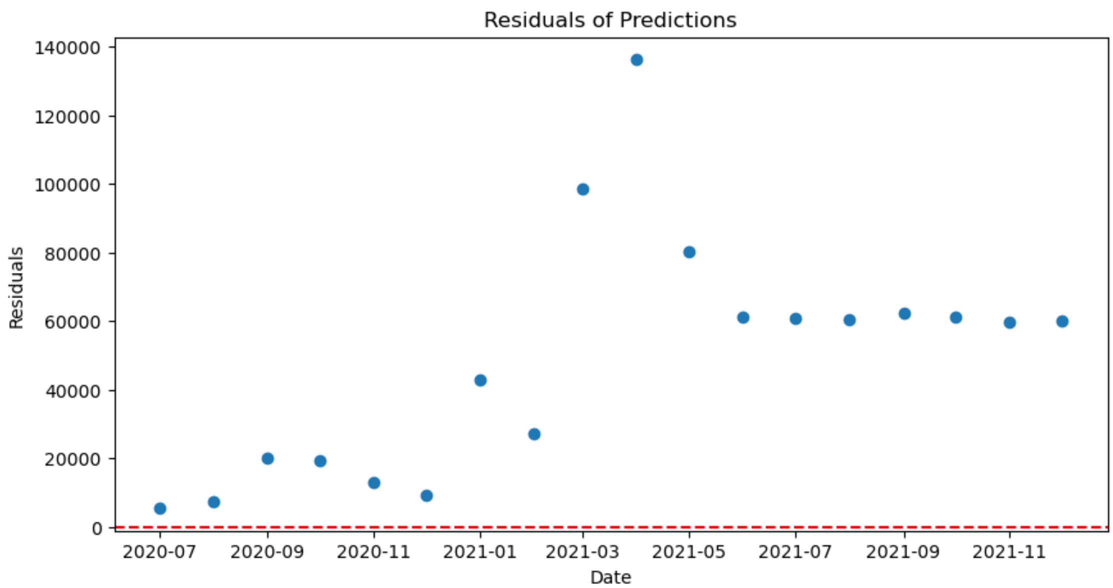
```
plt.fill_between(test.index, forecast_ci.iloc[:, 0], forecast_ci.iloc[:, 1],␣
↪color='pink', alpha=0.3)
plt.title('SARIMA Model Forecast vs Actual')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.show()
```

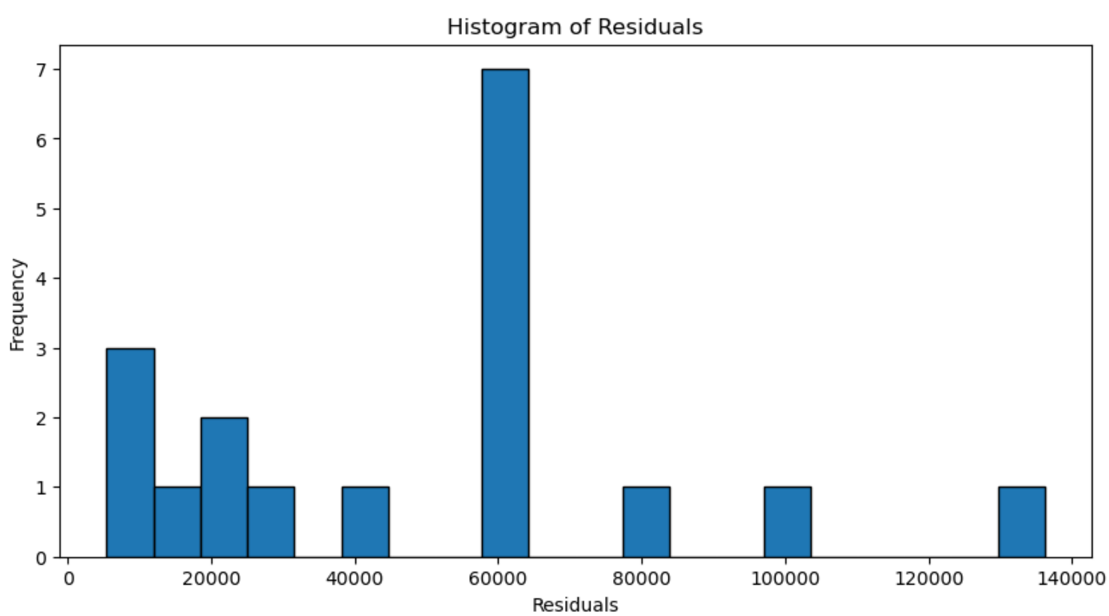


[35]:
```
# evaluating and plotting residuals
rmse = np.sqrt(mean_squared_error(test['Sales'], forecast_mean))
mae = mean_absolute_error(test['Sales'], forecast_mean)
mape = np.mean(np.abs((test['Sales'] - forecast_mean) / test['Sales'])) * 100
print("Root Mean Squared Error (RMSE):", rmse)
print("Mean Absolute Error (MAE):", mae)
print("Mean Absolute Percentage Error (MAPE):", mape, "%")

# calculating residuals
residuals = test['Sales'] - forecast_mean
plt.figure(figsize=(10, 5))
plt.scatter(test.index, residuals)
plt.axhline(y=0, linestyle='--', color='red')
plt.title('Residuals of Predictions')
plt.xlabel('Date')
plt.ylabel('Residuals')
plt.show()

# histogram of residuals
```

```python
plt.figure(figsize=(10, 5))
plt.hist(residuals, bins=20, edgecolor='black')
plt.title('Histogram of Residuals')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.show()
```

```
Root Mean Squared Error (RMSE): 59800.722130166985
Mean Absolute Error (MAE): 49211.94484200395
Mean Absolute Percentage Error (MAPE): 9.037681019809652 %
```





SARIMA Iteration One Interpretation This model looked like an improvement over the Holt model. The residuals were still sizable, but there was a marked reduction, so it seemed there was better alignment to the actual data trends.
I wanted to see a reduction in MAPE, as rough estimates are rarely appreciated for decision-making and planning when sales revenues are involved.
I tried a second iteration of the SARIMA below, this time performing a grid search to find the best hyperparameters for my model.

0.0.7 SARIMA Model – Iteration Two

One consideration I would make in future projects would be to fully understand the level of accuracy needed by the stakeholders.
In healthcare, for example, analytics used for retrospective quality initiatives can have an error rate of 5 - 9% without much concern. Analytics used to provide guidance at the point of care to live patients in real time must have a significantly higher match rate. My only exposure to finance has been in the realm of mid-stage, for-profit startups where inaccurate forecasting can very quickly lead to failure and business closure. This perspective may be limited and I planned to make additional Teams posts on it this week to explore other perspectives.

[38]:
```
# running auto_arima to find the best model
best_model = auto_arima(train['Sales'], start_p=0, start_q=0, max_p=3, max_q=3,␣
 ↪m=12,
                        start_P=0, seasonal=True, d=None, D=1, trace=True,
                        error_action='ignore',  # I don't want to know if an␣
 ↪order doesn't work
                        suppress_warnings=True,  # I don't want convergence␣
 ↪warnings
                        stepwise=True)

# summarizing the best SARIMA model
print(best_model.summary())
```

```
Performing stepwise search to minimize aic
 ARIMA(0,0,0)(0,1,1)[12] intercept   : AIC=7134.870, Time=0.25 sec
 ARIMA(0,0,0)(0,1,0)[12] intercept   : AIC=7133.130, Time=0.03 sec
 ARIMA(1,0,0)(1,1,0)[12] intercept   : AIC=6911.121, Time=0.50 sec
 ARIMA(0,0,1)(0,1,1)[12] intercept   : AIC=6984.435, Time=0.22 sec
 ARIMA(0,0,0)(0,1,0)[12]             : AIC=7331.835, Time=0.02 sec
 ARIMA(1,0,0)(0,1,0)[12] intercept   : AIC=7002.537, Time=0.07 sec
 ARIMA(1,0,0)(2,1,0)[12] intercept   : AIC=6973.369, Time=0.57 sec
 ARIMA(1,0,0)(1,1,1)[12] intercept   : AIC=6949.394, Time=0.37 sec
 ARIMA(1,0,0)(0,1,1)[12] intercept   : AIC=6904.308, Time=0.44 sec
 ARIMA(1,0,0)(0,1,2)[12] intercept   : AIC=6962.834, Time=0.51 sec
 ARIMA(1,0,0)(1,1,2)[12] intercept   : AIC=6940.168, Time=0.99 sec
 ARIMA(2,0,0)(0,1,1)[12] intercept   : AIC=6964.400, Time=0.25 sec
```

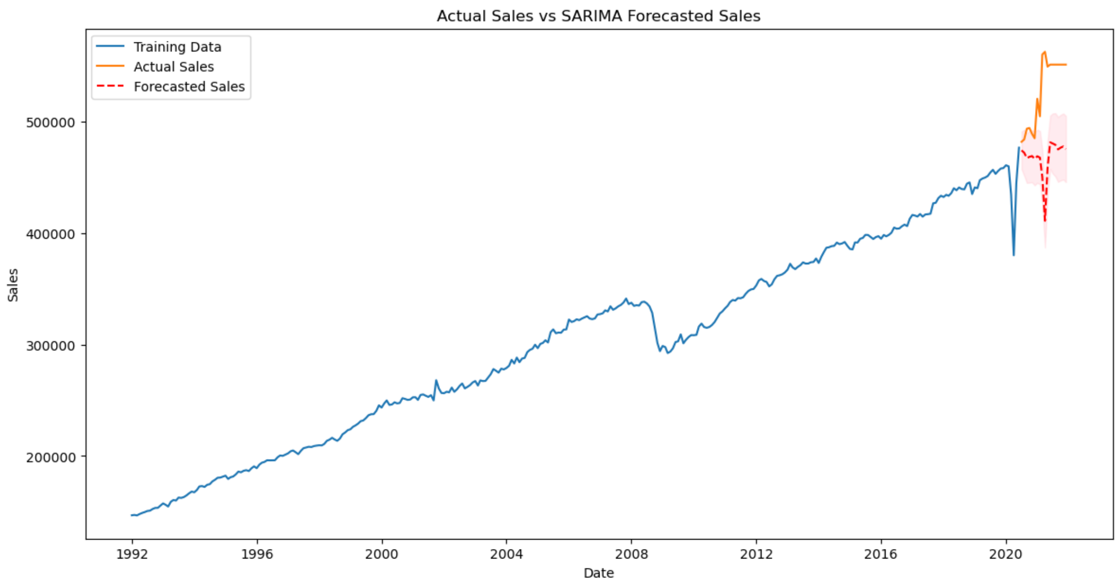
```
 ARIMA(1,0,1)(0,1,1)[12] intercept   : AIC=6904.967, Time=0.69 sec
 ARIMA(2,0,1)(0,1,1)[12] intercept   : AIC=6942.945, Time=0.64 sec
 ARIMA(1,0,0)(0,1,1)[12]             : AIC=6957.370, Time=0.42 sec

Best model:  ARIMA(1,0,0)(0,1,1)[12] intercept
Total fit time: 5.972 seconds
                                      SARIMAX Results
================================================================================
============
Dep. Variable:                                    y   No. Observations:
342
Model:             SARIMAX(1, 0, 0)x(0, 1, [1], 12)   Log Likelihood
-3448.154
Date:                              Fri, 03 May 2024   AIC
6904.308
Time:                                      17:35:21   BIC
6919.504
Sample:                                  01-01-1992   HQIC
6910.369
                                       - 06-01-2020
Covariance Type:                                opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept   2712.4794    428.513      6.330      0.000    1872.609    3552.350
ar.L1          0.7119      0.013     54.029      0.000       0.686       0.738
ma.S.L12      -0.2742      0.030     -9.002      0.000      -0.334      -0.214
sigma2      7.324e+07      0.032   2.31e+09      0.000    7.32e+07    7.32e+07
================================================================================
===
Ljung-Box (L1) (Q):                   8.49   Jarque-Bera (JB):
19382.00
Prob(Q):                              0.00   Prob(JB):
0.00
Heteroskedasticity (H):               0.90   Skew:
-3.95
Prob(H) (two-sided):                  0.56   Kurtosis:
39.70
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
[2] Covariance matrix is singular or near-singular, with condition number
6.98e+24. Standard errors may be unstable.
```

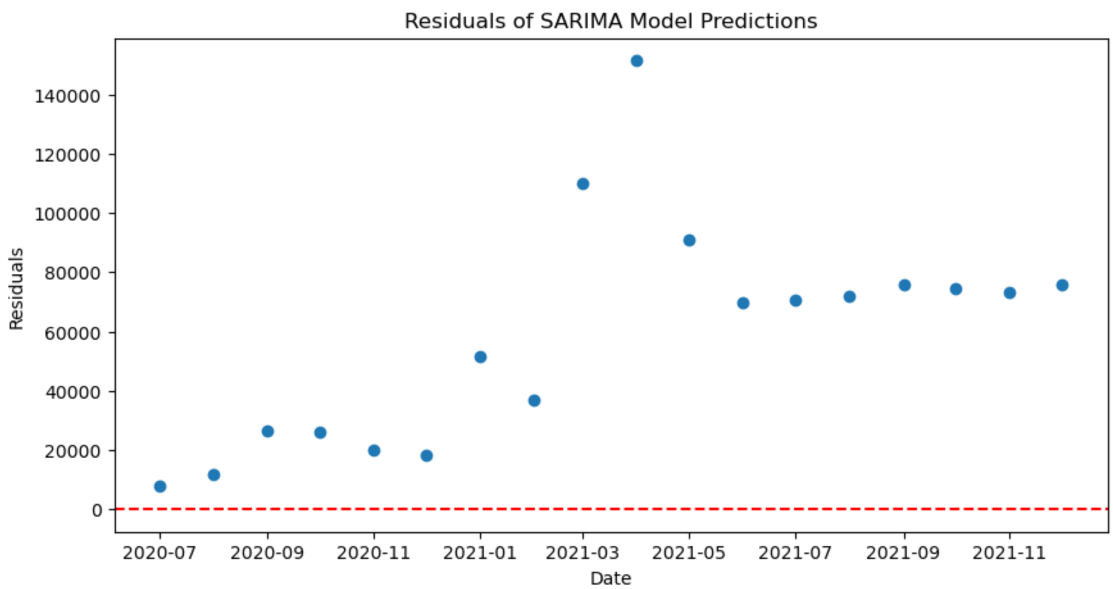
[39]:
```
# fitting the best SARIMA model
model = SARIMAX(train['Sales'], order=(1, 0, 0), seasonal_order=(0, 1, 1, 12),␣
 ↪trend='c')
results = model.fit()

# forecasting
forecast = results.get_forecast(steps=len(test))
forecast_mean = forecast.predicted_mean
forecast_ci = forecast.conf_int()
```

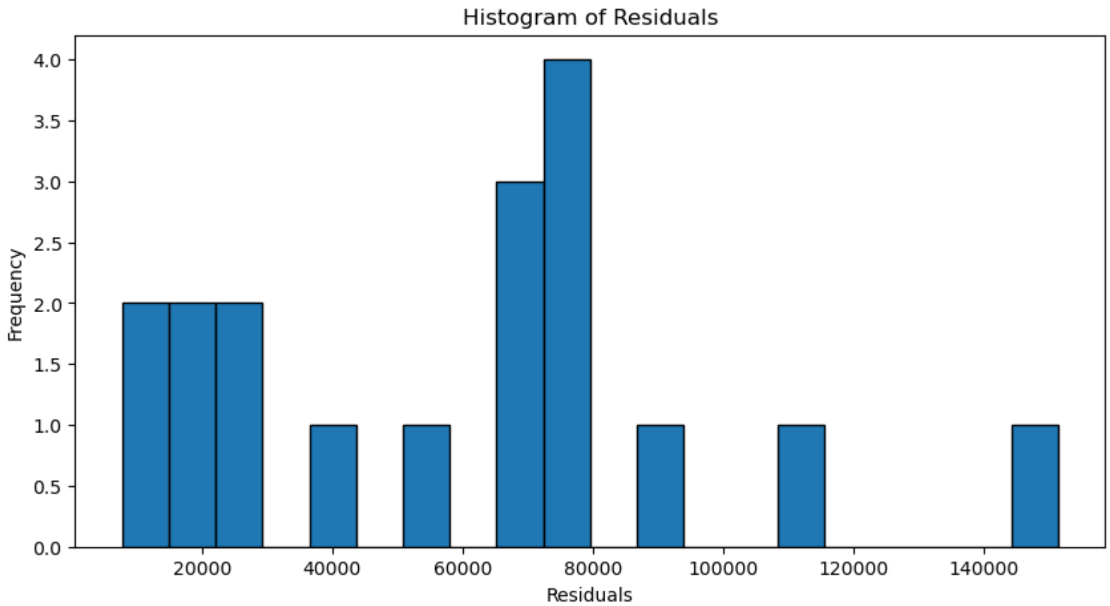
[40]:
```
# evaluation diagnostics
rmse = np.sqrt(mean_squared_error(test['Sales'], forecast_mean))
mae = mean_absolute_error(test['Sales'], forecast_mean)
mape = np.mean(np.abs((test['Sales'] - forecast_mean) / test['Sales'])) * 100
print("Root Mean Squared Error (RMSE):", rmse)
print("Mean Absolute Error (MAE):", mae)
print("Mean Absolute Percentage Error (MAPE):", mape, "%")

# plotting actual vs forecasted results
plt.figure(figsize=(14, 7))
plt.plot(train.index, train['Sales'], label='Training Data')
plt.plot(test.index, test['Sales'], label='Actual Sales')
plt.plot(test.index, forecast_mean, label='Forecasted Sales', color='red',␣
 ↪linestyle='--')
plt.fill_between(test.index, forecast_ci.iloc[:, 0], forecast_ci.iloc[:, 1],␣
 ↪color='pink', alpha=0.3)
plt.title('Actual Sales vs SARIMA Forecasted Sales')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.show()

# calculating residuals
residuals = test['Sales'] - forecast_mean
plt.figure(figsize=(10, 5))
plt.scatter(test.index, residuals)
plt.axhline(y=0, color='red', linestyle='--')
plt.title('Residuals of SARIMA Model Predictions')
plt.xlabel('Date')
plt.ylabel('Residuals')
plt.show()

# histogram of residuals
plt.figure(figsize=(10, 5))
plt.hist(residuals, bins=20, edgecolor='black')
plt.title('Histogram of Residuals')
plt.xlabel('Residuals')
```

```
plt.ylabel('Frequency')
plt.show()
```

```
Root Mean Squared Error (RMSE): 69524.99602098805
Mean Absolute Error (MAE): 59018.586212652765
Mean Absolute Percentage Error (MAPE): 10.873497172905921 %
```







0.0.8 Removing Outliers in the Test Data

As a final iteration, I re-ran the outlier detection methods near the top of this notebook, this time, focusing solely on the data used for testing.
While I didn't initially want to remove outliers, preferring a more conservative approach, I felt it was time to try one last adjustment, re-run the search for the best_model, and make a final attempt at an improved SARIMA model.

[46]:
```
# displaying the original value to confirm
print("Original data at 2021-01:", df_long.loc['2021-01'])

# setting the outlier value to NaN
df_long.loc['2021-01', 'Sales'] = np.nan

# checking
print("Data after setting outlier to NaN at 2021-01:", df_long.loc['2021-01'])

# interpolating to fill NaN values
df_long['Sales'] = df_long['Sales'].interpolate(method='linear')

# confirming change
print("Data after interpolation at 2021-01:", df_long.loc['2021-01'])
```

```
Original data at 2021-01:                 Sales
2021-01-01  494620.0
Data after setting outlier to NaN at 2021-01:              Sales
2021-01-01    NaN
Data after interpolation at 2021-01:                Sales
```

```
2021-01-01  494620.0
```

Re-running the Auto ARIMA to Find the Best Model, Post-Outlier Removal

[47]:
```python
# re-splitting the data, just to be sure
train = df_long[df_long.index < '2020-07-01']
test = df_long[df_long.index >= '2020-07-01']

# running auto_arima to find the best model
best_model = auto_arima(train['Sales'], start_p=0, start_q=0, max_p=3, max_q=3,␣
 ↪m=12,
                        start_P=0, seasonal=True, d=None, D=1, trace=True,
                        error_action='ignore',
                        suppress_warnings=True,
                        stepwise=True)

# summarizing the best SARIMA model
print(best_model.summary())
```

```
Performing stepwise search to minimize aic
 ARIMA(0,0,0)(0,1,1)[12] intercept   : AIC=7134.870, Time=0.24 sec
 ARIMA(0,0,0)(0,1,0)[12] intercept   : AIC=7133.130, Time=0.02 sec
 ARIMA(1,0,0)(1,1,0)[12] intercept   : AIC=6911.121, Time=0.57 sec
 ARIMA(0,0,1)(0,1,1)[12] intercept   : AIC=6984.435, Time=0.25 sec
 ARIMA(0,0,0)(0,1,0)[12]             : AIC=7331.835, Time=0.02 sec
 ARIMA(1,0,0)(0,1,0)[12] intercept   : AIC=7002.537, Time=0.07 sec
 ARIMA(1,0,0)(2,1,0)[12] intercept   : AIC=6973.369, Time=0.57 sec
 ARIMA(1,0,0)(1,1,1)[12] intercept   : AIC=6949.394, Time=0.38 sec
 ARIMA(1,0,0)(0,1,1)[12] intercept   : AIC=6904.308, Time=0.43 sec
 ARIMA(1,0,0)(0,1,2)[12] intercept   : AIC=6962.834, Time=0.50 sec
 ARIMA(1,0,0)(1,1,2)[12] intercept   : AIC=6940.168, Time=0.84 sec
 ARIMA(2,0,0)(0,1,1)[12] intercept   : AIC=6964.400, Time=0.23 sec
 ARIMA(1,0,1)(0,1,1)[12] intercept   : AIC=6904.967, Time=0.70 sec
 ARIMA(2,0,1)(0,1,1)[12] intercept   : AIC=6942.945, Time=0.65 sec
 ARIMA(1,0,0)(0,1,1)[12]             : AIC=6957.370, Time=0.40 sec

Best model:  ARIMA(1,0,0)(0,1,1)[12] intercept
Total fit time: 5.886 seconds
                                      SARIMAX Results
================================================================================
============
Dep. Variable:                                    y   No. Observations:
342
Model:             SARIMAX(1, 0, 0)x(0, 1, [1], 12)   Log Likelihood
-3448.154
Date:                              Fri, 03 May 2024   AIC
6904.308
Time:                                      18:07:43   BIC
6919.504
```

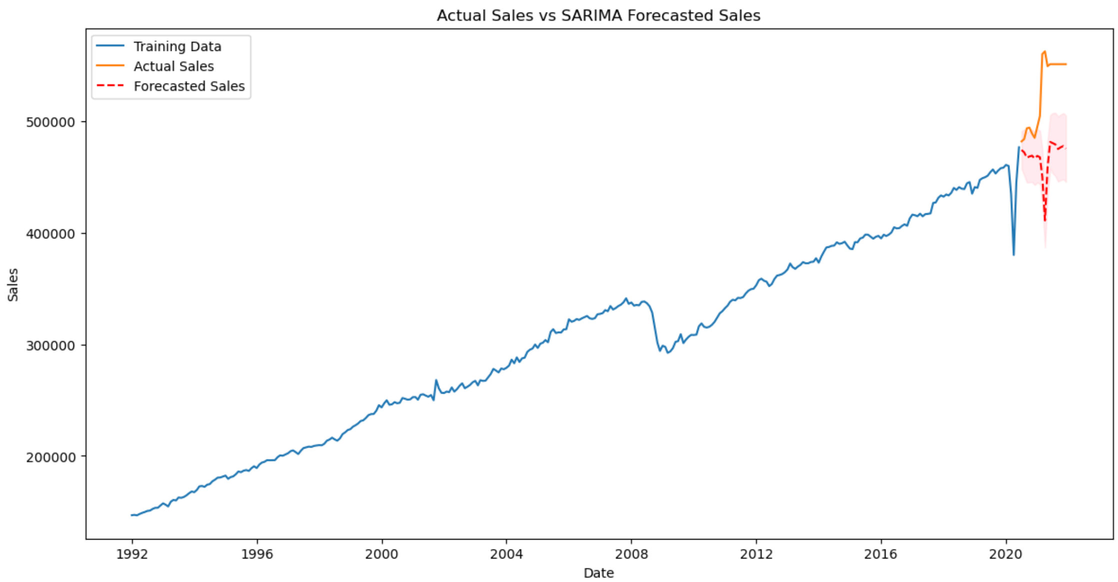
```
Sample:                               01-01-1992   HQIC
6910.369
                                    - 06-01-2020
Covariance Type:                             opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept   2712.4794    428.513      6.330      0.000    1872.609    3552.350
ar.L1          0.7119      0.013     54.029      0.000       0.686       0.738
ma.S.L12      -0.2742      0.030     -9.002      0.000      -0.334      -0.214
sigma2      7.324e+07      0.032   2.31e+09      0.000    7.32e+07    7.32e+07
================================================================================
===
Ljung-Box (L1) (Q):                  8.49   Jarque-Bera (JB):
19382.00
Prob(Q):                             0.00   Prob(JB):
0.00
Heteroskedasticity (H):              0.90   Skew:
-3.95
Prob(H) (two-sided):                 0.56   Kurtosis:
39.70
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
[2] Covariance matrix is singular or near-singular, with condition number
6.98e+24. Standard errors may be unstable.
```

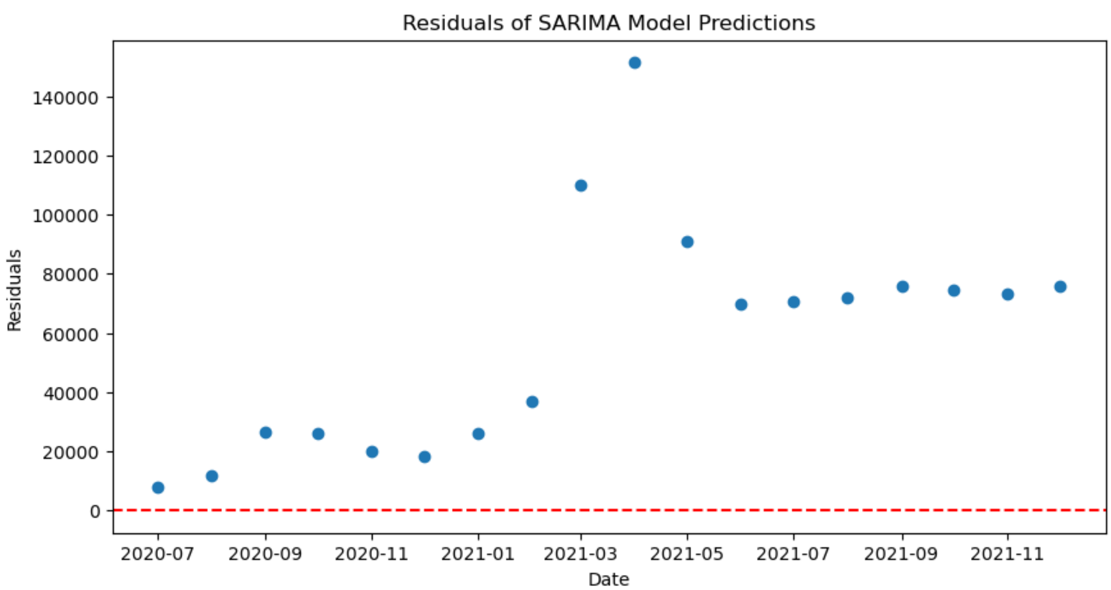
Results Interpretation The results of the auto-arima were identical before and after the removal and replacement of the outlier.
The chosen configuration may be robust enough that removing a single outlier didn't affect the overall model selection. This could be a good indication that my model is stable.
Another possibility is that my outlier simply didn't have a substantial influence on the overall parameters.
I chose to re-evaluate the same model to see if any diagnostic checks improved at all.

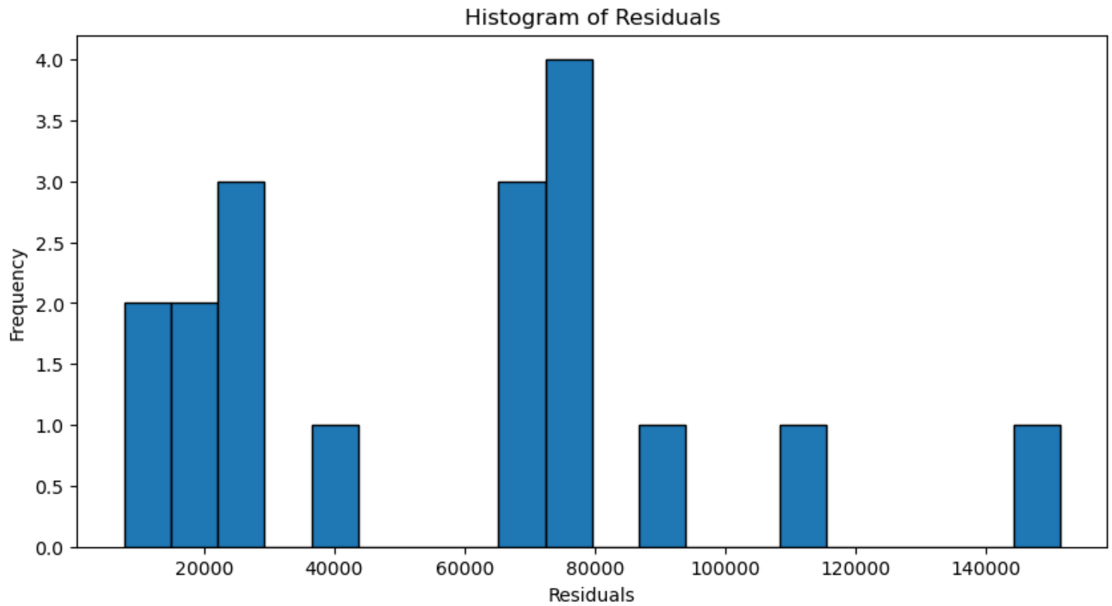
[49]:
```
# forecasting on test set
forecast = best_model.predict(n_periods=len(test))

# evaluation diagnostics
rmse = np.sqrt(mean_squared_error(test['Sales'], forecast_mean))
mae = mean_absolute_error(test['Sales'], forecast_mean)
mape = np.mean(np.abs((test['Sales'] - forecast_mean) / test['Sales'])) * 100
print("Root Mean Squared Error (RMSE):", rmse)
print("Mean Absolute Error (MAE):", mae)
```

```python
print("Mean Absolute Percentage Error (MAPE):", mape, "%")

# plotting actual vs forecasted results
plt.figure(figsize=(14, 7))
plt.plot(train.index, train['Sales'], label='Training Data')
plt.plot(test.index, test['Sales'], label='Actual Sales')
plt.plot(test.index, forecast_mean, label='Forecasted Sales', color='red',␣
 ↪linestyle='--')
plt.fill_between(test.index, forecast_ci.iloc[:, 0], forecast_ci.iloc[:, 1],␣
 ↪color='pink', alpha=0.3)
plt.title('Actual Sales vs SARIMA Forecasted Sales')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.show()

# calculating residuals
residuals = test['Sales'] - forecast_mean
plt.figure(figsize=(10, 5))
plt.scatter(test.index, residuals)
plt.axhline(y=0, color='red', linestyle='--')
plt.title('Residuals of SARIMA Model Predictions')
plt.xlabel('Date')
plt.ylabel('Residuals')
plt.show()

# histogram of residuals
plt.figure(figsize=(10, 5))
plt.hist(residuals, bins=20, edgecolor='black')
plt.title('Histogram of Residuals')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.show()
```

```
Root Mean Squared Error (RMSE): 68729.8399823039
Mean Absolute Error (MAE): 57599.58621265276
Mean Absolute Percentage Error (MAPE): 10.615018391167567 %
```


Actual Sales vs SARIMA Forecasted Sales
Training Data
Actual Sales
Forecasted Sales
Sales
500000
400000
300000
200000
1992
1996
2000
2004
2008
2012
2016
2020
Date


Residuals of SARIMA Model Predictions
Residuals
140000
120000
100000
80000
60000
40000
20000
0
2020-07
2020-09
2020-11
2021-01
2021-03
2021-05
2021-07
2021-09
2021-11
Date



0.0.9 Conclusion

The Holt's Linear Trend model ended up being the best one of all these iterations, in my opinion. Here were the evaluation diagnostics of that method again:

Root Mean Squared Error (RMSE): 52946.873413229114

Mean Absolute Error (MAE): 45474.2090290907

Mean Absolute Percentage Error (MAPE): 8.361778411333004

It may be an indication that sometimes the simplest fit-to-data approach is the best.
It may be that I am not close enough to consider this model "good enough". Without more knowledge of stakeholder expectations, however, I can't truly evaluate if this is accurate enough for the requestors.

I am hoping to learn more in discussions about this assignment this week as I found this assignment quite challenging. Thank you for any input on my thinking process here you can provide as well!
%

0.0.10 References

Dean Abbot, (2014). Applied Predictive Analytics: Principles and Techniques for the Professional Data Analyst. Indianapolis, IN: Wiley.

Wes McKinney, (2022) Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter, 3rd ed. Sebastopol, CA: O'Reilly

Python Software Foundation. Python Language Reference, version 3.9. Available at [1][1].

Luna, Fernando (2023). Medium, “Time Series Preprocessing” URL: https://medium.com/@lu.fernando2901/time-series-preprocessing-946ccaff3ff1

Ariton, Lleyton (2023). Analytics Vidha, “A thorough Introduction to Holt-Winters Forecasting” URL: https://medium.com/analytics-vidhya/a-thorough-introduction-to-holt-winters-forecasting-c21810b8c0e6

Brownlee, Jason (2019). Machine Learning Mastery, “A Gentle Introduction to SARIMA for Time Series Forecasting in Python” URL: https://machinelearningmastery.com/sarima-for-time-series-forecasting-in-python/

[ ]:

```

```