UNIQX Gift Sales Forecast Project

John Chen, Azucena Faus

Shiley-Marcos School of Engineering, University of San Diego

Abstract

Online retailers need to have a quantitative look at their sales data to determine future inventory needs and profit margins. This is because an online store lacks the human intuition (from say, sales associates) that can be gathered from a brick-and-mortar store. A special case arises from specialty item shops that cater to a distinctive clientele, as in the unique-gift online retailer, UNIQX. Trends shift and change constantly, and these shops often sell elusive versions of trending items to provide the best shopping experience for their clients. A study on a particular item that has growing interest was conducted and a 30-day forecast of sales was acquired based on 12 months of available retail data from the company. Various data-driven methods were used along with a few model-based methods. Surprisingly, the model-based methods did not outperform some of the best data-driven models such as Holt-Winters (Triple Exponential Smoothing) Method. This study confirms that although more complex models often help in cross-sectional data prediction scenarios, this may not always be the case with time series forecasting. If improvements are vital and the business needs require better performance, perhaps a more complex version of such models (or those readily available in the R Studio platform) may have a shot in providing better results.

Keywords: machine learning, time series, time series forecasting, naïve forecast, autocorrelation, regression, arima, neural network forecasting, long short-term memory, LSTM, exponential smoothing

Table of Contents

Contents

Time Series Forecast of projected Sales for UNIQX Online Giftshop

UNIQX is a UK-based online gift retailer that specializes in unique gifts and also sells to wholesale clients. Although the store ships internationally, most sales are in the UK.

Introduction

United Kingdom has one of the most internet-savvy populations in the world, where about 96% of the population has internet access (United Kingdom - Ecommerce. International Trade Administration). Over the past couple years, there has been an increase in online sales, and this has been projected to continue to increase. Therefore, it is imperative that online retailers look at their data closely and create algorithms to forecast future sales to make sure there is enough inventory for customers.

Methods for conducting sales forecasting are varied, including data-driven methods like autoregression, moving averages, and smoothing methods. There are also some model-based methods like linear regression and neural networks that are also used for forecasting.

Literature Review

E-commerce sales have grown exponentially over the last decade and even more so now due to the pandemic with many physical stores shutting down. Internet shopping in the United Kingdom is more popular than any other country, where eCommerce is 30% of the retail market and 82% of the population has bought at least one item online in 2021 (United Kingdom - Ecommerce. International Trade Administration). Some of the benefits of online shopping are: 1) ease of shopping with a click of a button on phones, 2) saves time without having to drive to the store, 3) contactless, and 4) includes the ability to browse for the best deal. Interestingly, as convenient as online shopping is, using the phone for all things, such as morning alarm functions, can have a negative impact on people's morning routines

(Bumpus, 2022). This has propelled a come-back of the traditional alarm clock. Advantages of the alarm clock are obvious; it tells the time, has an alarm, can be portable, and more importantly, reduce people's reliance on phones to start off their day.

Due to the growing trend in clock purchases, the focus for this study was on the sales of clocks for UNIQX's time series sales data. Some accepted norms for working with time series data such as this are regression models as well as some data-driven methods. In the study *Machine-Learning Models for Sales Time Series Forecasting. Data Mining and Processing* by Bohdan M. Pavlyshenko, they mention that Sales forecasting is more of a regression problem than a time series problem. The reason for this is because usually machine learning models assume global trends (what happened in the past will happen again) and hence get better results than traditional time series forecast algorithms like smoothing and ARIMA. Therefore, although time series methods will be explored as baseline models, other machine learning algorithms will also be explored and compared to find out if best results come from such models.

Problem Definition

UNIQX needs a sales forecasting algorithm for an in-demand specialty gift item that is on trend. The data science team at UNIQX is to provide the best model and results that predicts the next 30 days of sales for such an item. It has been determined that the on-trend gift item of interest as of late that is on the rise are clocks. Hopefully this algorithm can later be used to forecast trends in other specific items as trends change. The largest error that the business can reasonably sustain is +/-200£ (RMSE 200), given the mean sales of clocks per day was about 475£, while the range of sales of clocks per day often reached to the thousands of Pounds (£).

Exploratory Data Analysis and Pre-Processing

The data provided by UNIQX contains 541,909 records where each record represents a transaction.

Data Cleaning

The dataset contained all transactions relevant to the business, so data cleaning was needed to analyze real purchase transactions for the item of interest versus the rest of many transactions. Each transaction had 9 variables: 'InvoiceNo,' 'StockCode,' 'Description' (the item description), 'Quantity' (amount purchased), 'InvoiceDate,' 'UnitPrice,' 'CustomerID,' and 'Country' (from which order originated).

Missing/Irrelevant Data

The dataset had some missing values, but none of which affected the variables that dealt with the sales for each transaction. 'UnitPrice' and 'Quantity' were available for each record, which were used to calculate 'Sales', the outcome variable of interest. However, there were some house-keeping records that kept track of postage fees, amazon fees, as well as returns. These records had to be sifted out from the data so only purchase transactions (that were never returned/refunded) remained.

Exploratory Data Analysis

Some observations gathered from the dataset involved finding the most popular items sold, the range of sizes in orders based on quantity (up to over 4,000), from many parts of the world. Most sales originated from the United Kingdom, which is where UNIQX is based. The Unit Price of items did not seem to deter any sized order, as they span the price range.

The time series was also observed as a whole, as well as based on order sizes small (less than 100), medium (between 100 and 1000), and large (>1000). Most orders fell into the small category with

roughly 513,000 rows, medium had just over 6,200 rows, and large orders were only 102 rows. Bar graphs were created to observe the top 10 items for each group, and most were miscellaneous items such as t-light holders, cake stand, retrospots, bunting, cake cases, chalkboards, and ornaments. The top item was t-light holders with 2,293 customer orders. However, when looking at total sales for that item, it did not compare with the sales from clocks.

The histograms were created for the small, medium, and large orders which showed both 'Quantity' and 'UnitPrice' histograms being skewed to the right. The most popular item ordered 'White Hanging Heart T-Light Holder' had total sales of just over 94,000£ from December 2010 to December 2021. The team looked at various items being sold and noticed there were quite a few related to clocks and decided to look into that further. The total number of clock orders from customers were 7,025 with most clocks coming from small orders and only a few in the medium category size. Since there has been a growing interest in clocks, the team decided to look into the sales of clocks for creating a forecast. The total sales were around 168,000£ for all clocks, performing much better than the top selling item.

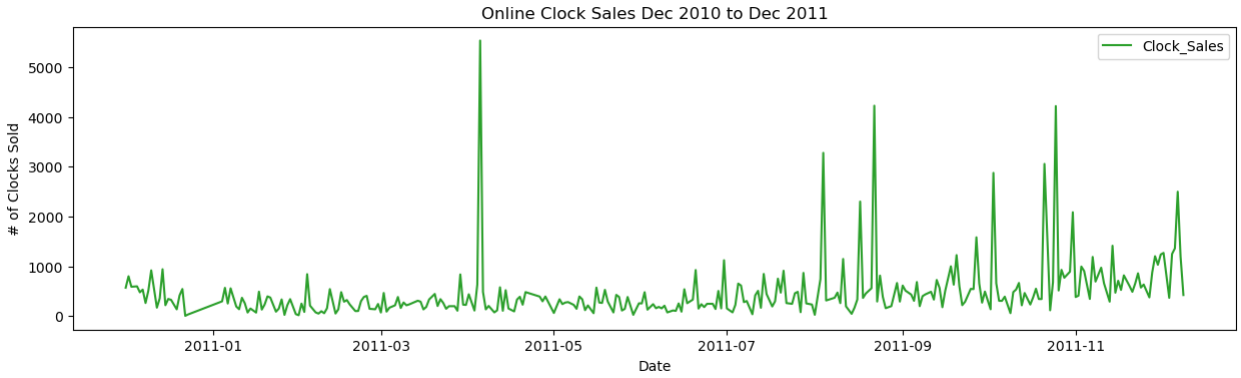
Another benefit for choosing clocks is that they are not a 'random' item which appears to be the case for the top selling items. Due to those reasons, the team decided to base the time series project on clock sales. Other EDA that completed for the clock data was a scatterplot showing sales by country and number of clocks sold and the overwhelming majority were purchased in the United Kingdom giving the team additional focus to look solely at clock sales in the United Kingdom.

Specifically, the rows with clock sales were selected from the data. 'InvoiceDate' was converted to datetime attribute and only UK orders were selected from the 'Country' column. For the purposes of creating time series sales forecast, the team removed the columns 'InvoiceNo', 'CustomerID', 'UnitPrice', 'Quantity', 'StockCode', 'Description', and 'Country'. The final dataframe only contained 'InvoiceDate' and 'Sales' for clocks from December 2010 to December 2011 and was ready for the team to use for modeling. Because there were various types of clocks and multiples orders per date, the data

was resampled using day and clock sales aggregated such that only days that had sales were included in the data frame. The only days that had no sales were every Saturday of every week, which is when the site does not record orders (the "store" is for all intents and purposes, closed). Figure 1 below shows the initial time series plot with daily sales of clocks.

Figure 1

Time Series for Clock sales



Since it was found that this time series is stationary and not a random walk, models were trained to forecast for clock sales. This way, the business can project inventory needs as well as profits/margins for the upcoming months.

Data Preprocessing for Modeling

Time series data differs from cross-sectional data in that the sequence of the data must be preserved. There are also some forecast methods that require uniform time intervals between records, which is why analysis was done on a daily sales condition. Some models do not have the regular time interval requirement and can work with data that has missing time intervals, like regression, so in those cases, missing values can be forecasted, imputed, then included in the final dataset to train the model. Finally, most models require that a time series be considered stationary; that is, that the time series

does not exhibit neither trend nor seasonality. To remove these systemic characteristics from the time series, a mathematical operation of differencing is usually implemented.

In the case of the dataset used, since the item of interest was clocks, and country of interest was United Kingdom, the dataset was narrowed down to these two categories. Subsequently, a new feature was engineered called 'Sales' which is the product of 'UnitPrice' and 'Quantity,' to quantify the total Sales for each transaction for clocks in the UK. Additionally, the data was aggregated to daily sales and days in which the site is closed (which had 0 Sales values) were removed since the site is only active 6 days a week (Sunday-Friday).

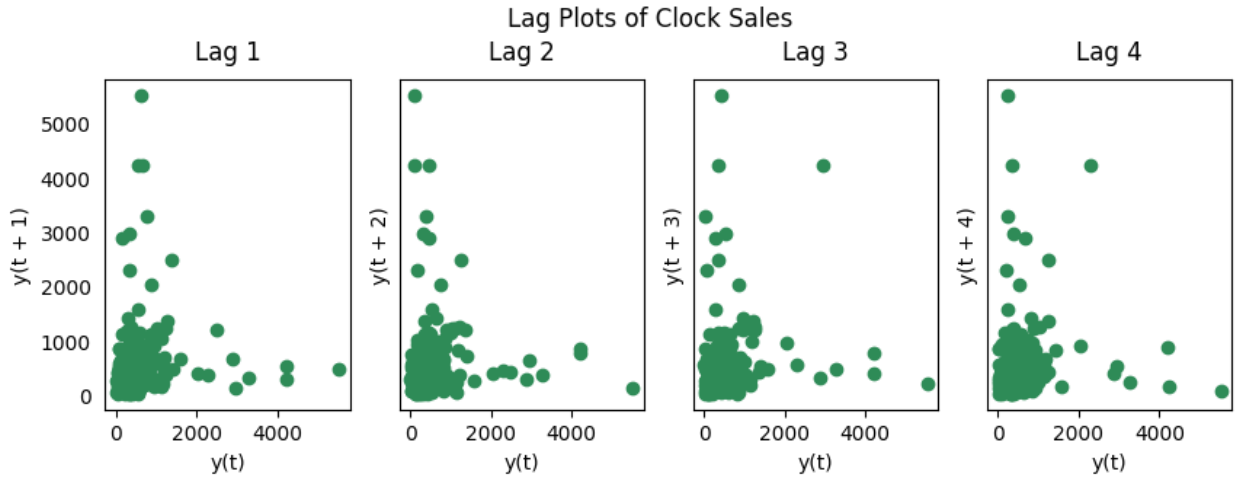
In addition, the team took a look at the trend, seasonality, and residuals using two decomposition methods additive and multiplicative. Trend is the long-term direction of the time series which can be increasing, decreasing or constant. Seasonality is a periodic behavior of the time series within a period of a year. Residuals is the remainder of the time series after trend and seasonality are removed. Looking at the additive decomposition results we see that the residuals have some pattern left over, but with multiplicative decomposition the residuals look quite random which is ideal and therefore the team decided to go with multiplicative and the time series was well decomposed.

Afterward, the stationarity of the time series was evaluated by means of the Augmented Dickey-Fuller test function available in the statsmodels package. The null hypothesis for the ADF test is that the signal is non-stationary, and a p value less than 0.01 indicated the null hypothesis could be rejected, and hence the time series was stationary, fulfilling the requirement for modeling. As to the predictability of the series, an autoregression of lag-1 resulted in a coefficient much less than 1. This confirmed the signal was not a random walk, and was therefore predictable (Shmueli & Lichtendahl Jr., 2018). In the case of neural network modeling, the data had to be further processed before splitting and was scaled to be between 0 and 1 (the inverse of the scaling had to be applied once predictions were collected, for proper error calculations).

Lag plots are scatter plots of a time series against a lag of itself. They are used to check if the time series is random white noise. The team created four graphs up to lag 4 and if there are any patterns that exists in these graphs the time series can be confirmed to be autocorrelated. As seen in Figure 2, all four lag graphs have a mostly positive correlation pattern. Thus, the dataset is not random white noise.

Figure 2

Lag Plots to check for autocorrelation



To preserve the sequence of data, splits for training and validation datasets were done on sequential blocks of the data. The data spanned from December 1, 2010 until December 9, 2011. The last 30 days of the time series was used as validation data on which to compare predictions for the models.

Modeling, Methods, Validation, and Performance Metrics

A collection of nine time series forecasting methods were adapted to predict future sales for the online retail giftshop. Metrics that are usually used to determine performance of time series forecasting methods are the Root mean Squared error (RMSE), Akaike information criterion (AIC), and mean absolute scaled error (MAPE). In this case, RMSE was the determinant for best model given the business

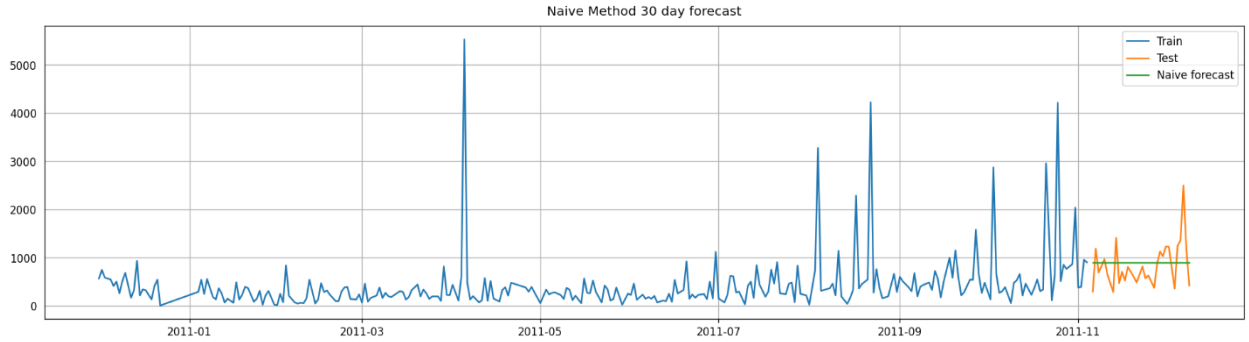
requirement from the stakeholders at UNIQX of a minimum profit loss that translates 1:1 with RMSE units and Sales, 200£ (200 RMSE).

Naïve (Persistence) Model

The persistence model is a data-driven method that uses the past value in the time series to predict the next event. This model is often considered "good enough" for many forecasting projects. At minimum, it should be applied as a baseline model to compare any other methods against. Only if other methods improve on the residuals from the naïve method should there be any consideration to move away from this simpler and explainable approach. After running the naïve forecast, the forecast line appeared to be flat and not suitable for data with high variability.

Figure 3

Naïve performance on known data

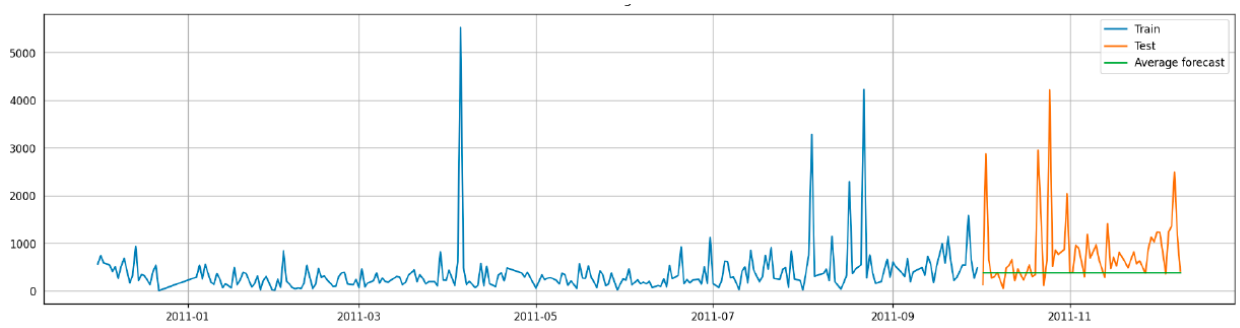


Simple Average Model

The simple average or mean model is the method where the forecast is based on the mean of the series based on time period *t.* This model does not take into account trend or seasonality. To the team's surprise this model actually performed almost as good as the ARIMA model based on the RMSE value.

Figure 4

Simple Average performance on known data

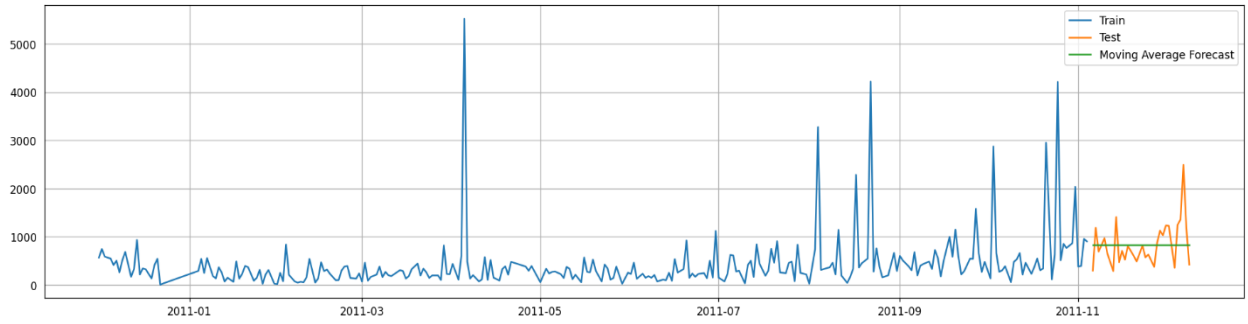


Moving Average Model

Moving average is a simple smoothing model, which is useful for forecasting time series data that does not show trend or seasonality. The moving average model is created by averaging values across a window of consecutive periods. The team used a 30-day moving average in the model because the sales patterns from the last 30 or so days of the training data seemed to closely match the 30 days of sales data of the test data. Using the 30-day moving average, the team achieved the lowest RMSE, but there are many models to go, and these results still may not get considered.

Figure 5

Simple Exponential Smoothing performance on known data



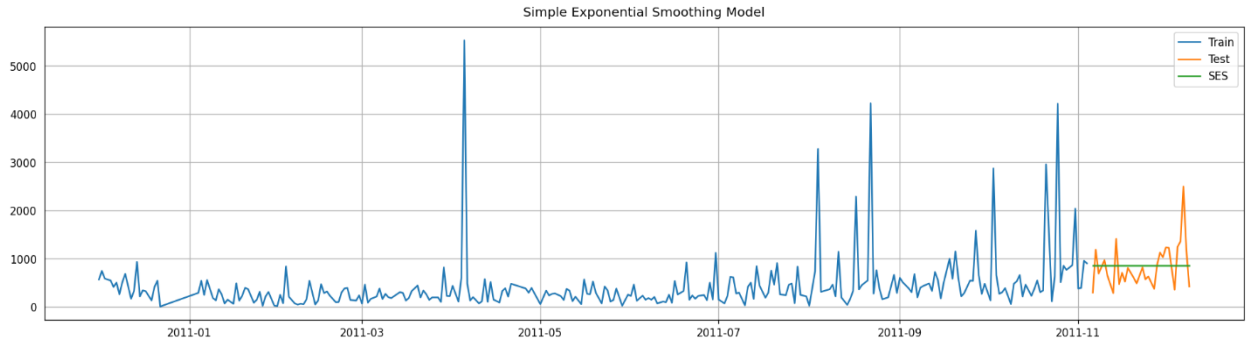
Simple Exponential Smoothing (SES) Model

Simple Exponential Smoothing is also a data-driven approach since past values (note the plural this time) are used to predict. In this case, a weighted average of the past values in a time series are

calculated. The weights give more importance to recent information and decrease exponentially into the past values. SES models use a smoothing constant, α that is defined by the user. The smoothing constant relates to how fast the model learns, the higher, the faster the learning. SES requires that the time series be stationary, that is, absent of trend and seasonality so these were removed before applying the model. Through trial and error, the alpha or smoothing constant was adjusted and a value of 0.6 was decided on which gave the best RMSE so far out of the models run. This model may be in consideration and discussed in the final results.

Figure 6

Simple Exponential Smoothing performance on known data

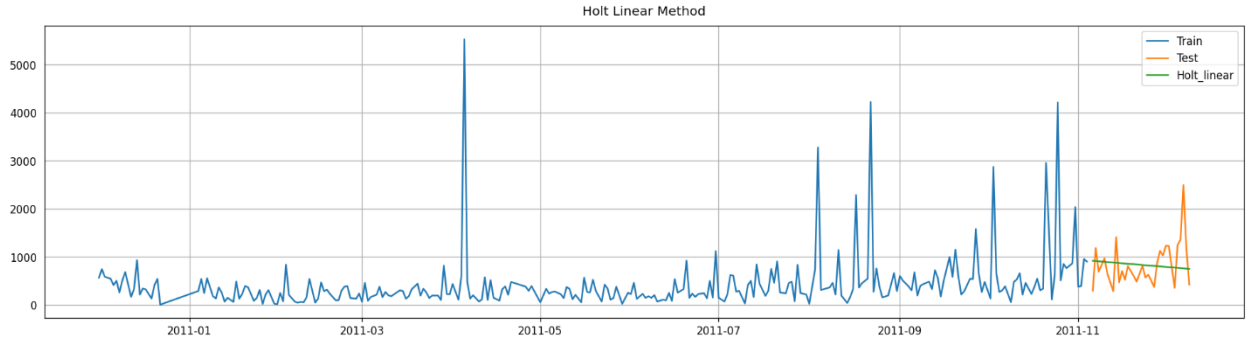


Holt's Linear Trend (Double Exponential Smoothing) Model

Holt's Linear trend model is an extension of the Simple exponential smoothing model, where the assumption of stationarity for the time series no longer holds. Holt's model allows for a time series with trend to be forecasted. Local trend is estimated from the data along with the level. This model has two user defined smoothing constants, α and β. Once again, the team used a trial-and-error method to determine the two smoothing constants and a smoothing level of 0.3 and slope of 0.1 were used in the model. The RMSE was further improved and these results will be further discussed.

Figure 7

Holt's Linear Trend performance on known data

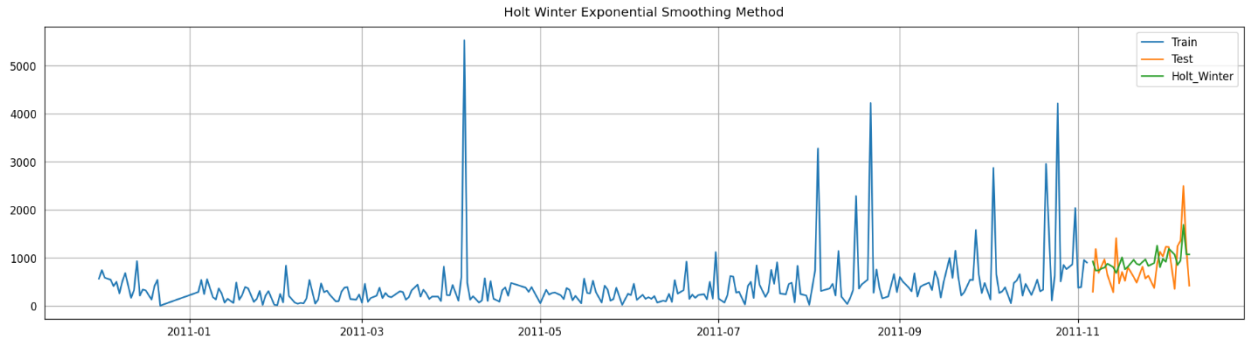


Holt-Winter's Exponential Smoothing Model

Holt-Winter's is a further extension of the Holt's linear trend model in that the time series can now have trend and seasonality for application. The smoothing constants for this model are α, β, and γ. The model was created using the ExponentialSmoothing function from statsmodels.tsa.api using the following parameters 'seasonal_period' of 29, 'trend of 'add', and seasonal as 'add', giving the best results and lowest RMSE out of all the models used. As you can see in Figure 8, the green forecast line does a much better job in following the movement of the test data.

Figure 8

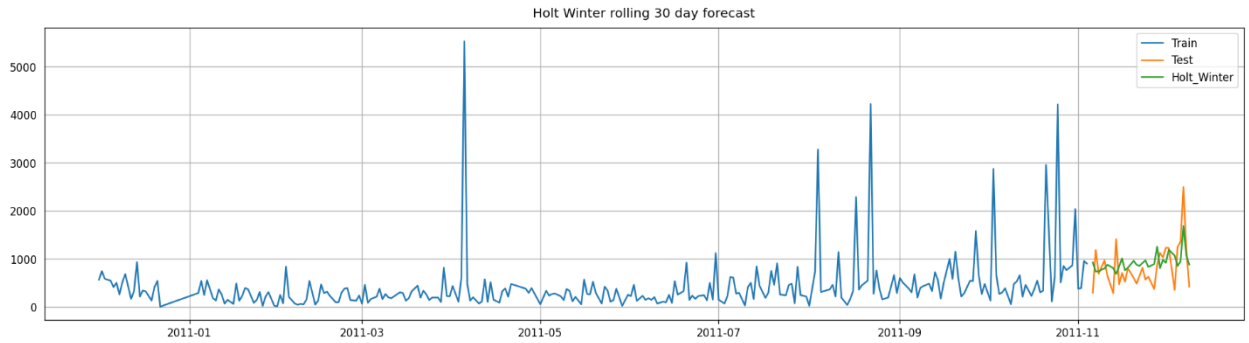
Holt-Winters' performance on known data



Another method was implemented with this model that iteratively updated the training data as every forecast was done one day at a time; including the new forecast in the training data to forecast yet the next day. Results were somewhat better, although perhaps not enough to merit the extra effort.

Figure 9

Holt Winter Iterative Daily 30-day forecast

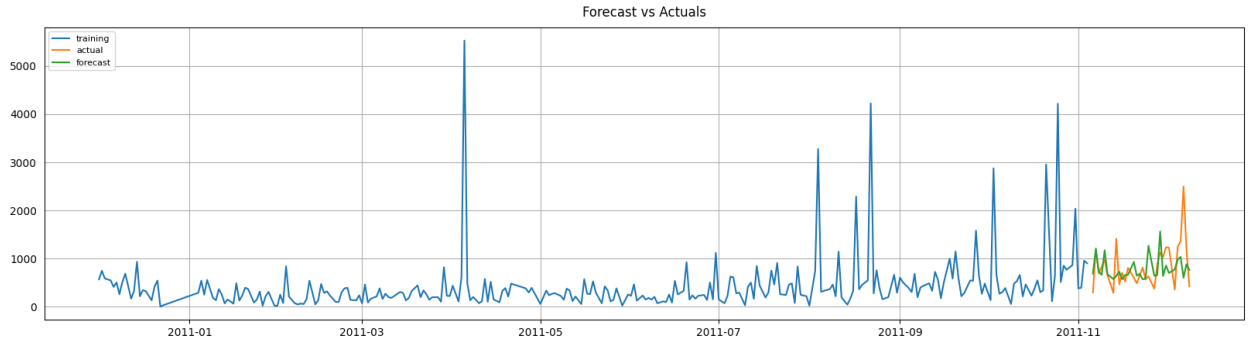


ARIMA Model

ARIMA models combine the benefits of an autoregressive model to predict forecasts using past values, a moving average to work with forecast errors to perfect future predictions, and differencing (d) to get rid of trends in the time series. The autoregression (AR) portion captures the autocorrelations of the series at p time intervals. The autocorrelation terms are then added to get the errors as a moving average for up to lag q. The constants (p, d, q) were determined by the user through trial and error and parameter tuning. After running various values for each constant, the team found that a p = 1, d = 0, and ma = 1 worked best and gave us our best ARIMA model with lowest AIC, BIC, and RMSE values. After adding seasonal parameters, AIC got even better (4265 vs 3855).

Figure 10

ARIMA performance (1,0,1)

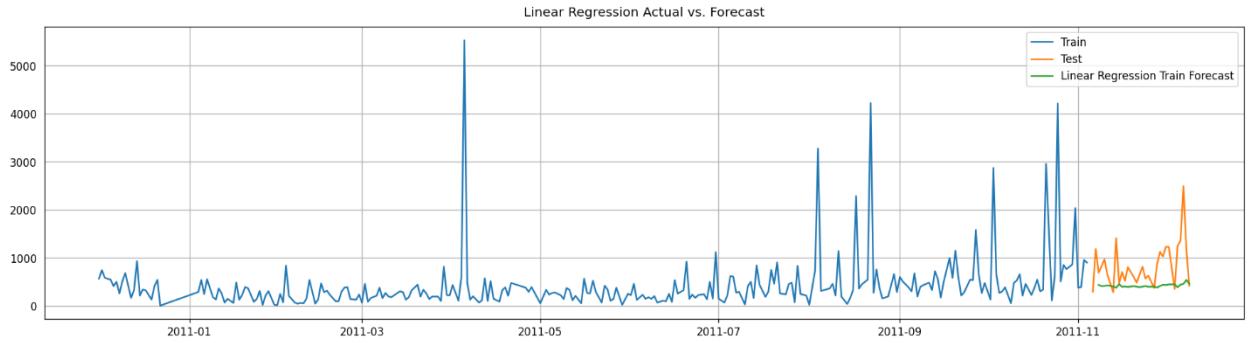


Linear Regression

Linear Regression is a machine learning algorithm that finds linear relationships between the input predictor variables and an outcome variable. In order to use this algorithm for time series forecasting, the lagged versions of the data is provided as input predictors, and the current data is the outcome that must be predicted. The initial model was tested with known test/validation lagged data to compare performance. Although the predictions followed the data, it still did not outperform the best model RMSE, so a rolled input version was not included.

Figure 11

Linear Regression performance on known data

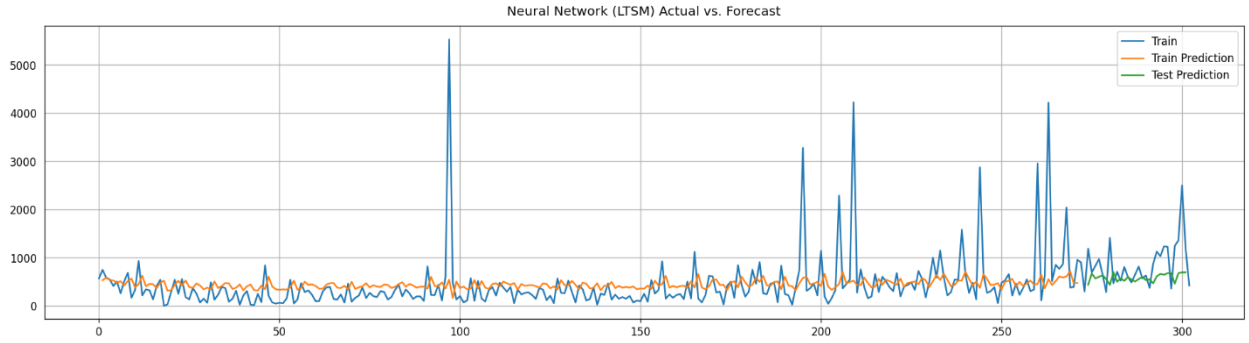


Neural Network

As mentioned earlier, neural networks are suggested as an effective forecasting method for time series sales predictions. Because there are no worries regarding client personal information, a block box approach should not violate the UK privacy laws. However, the complexity of the model may not merit any - if any - improvement in forecast. The model used was the Long Short-Term Memory Recurrent Neural Network. Data had to be scaled in this case and prepared in a matrix that specified the steps and number of features. In the end, the model did not outperform any other data-driven models in this application.

Figure 12

Neural Network LSTM performance



Modeling Results and Findings

The top three performing models were the Holt Winters, Holt Linear, and Moving Average. Value used to determine the best model was RMSE, which is the average error that a model's prediction has compared to the actual. For example, a RMSE score of 700 means the sales may be over by 700 on a given day. This can be a decent value given that sales were anywhere from 1-5000 per day. However, on a bad sales day the RMSE is still not very good. The Iterative Holt Winters method gave the lowest and best RMSE score out of all the models with a value of 392.78 followed by Holt Winter with 401.7, and

lastly Moving Average tied with Simple Exponential Smoothing with a score of 449.3. Other models were not considered since the RMSE scores were considerably higher than these three.

Table 1

Final RMSE Results for UNIQX Clock Retail Time Series

Model	RMSE
Holt Winter iterative	392.78
Holt Winter	401.7
Moving Average	449.3
Simple Exponential	449.3
Naive	452.17
ARIMA (1,0,1)(1,1,1,30)	470.62
Holt Linear	470.65
Neural Network (LSTM)	533.65
Simple Average	613.02
ARIMA (1,0,0)	613.18
Linear Regression	651.3

Final Model

While all three models performed similarly across RSME, other factors were considered such as how well the forecast fit the actual test data. Both the moving average and Simple Exponential Smoothing models forecasted what appeared to be flat lines, which were unlikely considering the increase in demand for clock orders. Holt Winters, on the other hand, gave a forecast that went up and down and seemed to follow the pattern of the actual data giving the team a more realistic outlook on sales performance. Therefore, the final model the team decided on is the Iterative Holt Winter Method, since in this project, the extra margin is needed.

Discussion of Results and Next Steps

The total daily sales average looked to be under 1,000 £. This means an RMSE score of 392.78 from our best model is not good enough because it has the potential to over forecast the total sales by

393£ or higher each day. The required threshold for RMSE was score of 200 or lower and although the team did not hit the target this time there was a key observation that there was an observable seasonality every 60 days but only for the last few months of the dataset. It might be possible that the trend was more recent than expected, which is characteristic of trends. Perhaps a few new wholesale clients are making the difference in recent months. An idea for future steps it to implement ensemble methods that obtain an average of RMSE scores. It may also make sense to focus on a new product category, as well as have better parameter tuning for the top performing models, especially the ones that had smoothing constants. Alternative time series' to consider would be an hourly or weekly perspective that may capture more seasonality. Furthermore, it might help to look at data from order size (small, medium, and large) perspective to differentiate between wholesale and private clients and better serve them. Finally, use of the RStudio platform that contains forecasting functions like autoregressive neural networks (nnetar) and linear regression that accepts season and trend as predictors can be explored for any possible improvements.

Conclusion

Alarm clocks are still alive and well and demand appears to be on the rise going towards an upward trend. The CNN article summed it up best that phones should be separate from alarms clocks as phones tend to keep a person in bed and the alarm clock will give a person a sense of control (Bumpus, 2022). With the upward demand in clocks as well as the rise in online shopping in the UK, the data is already looking more promising with less randomness showing signs of seasonality and even trend. The best performing model, the Iterative Holt Winters fell short of the target RMSE score of 200 or under this time around at 392, but there have already been discussions on the next steps in future projects such as using ensemble methods. The challenge was in that clock sales data starts off in a random fashion that does not follow the trends that show up in later months. This makes it difficult to forecast as there was only a year's worth of data to work with. The backup plan for other studies in the future is to look at

different products and order sizes to see if the time series models can predict more accurately on those datasets.

References

Shmueli, G., & Lichtendahl Jr., K. C. (2018). *Practical Time Series Forecasting with R Hands-On Guide*. Axelrod Schnall Publishers.

UCI Machine Learning Repository. (2015). [Online Retail Data Set] [Online_Retail.csv]. Dr Daqing Chen, Director: Public Analytics group. https://archive.ics.uci.edu/ml/datasets/Online+Retail

Bandara, K., et al. (2019, December 9). *Sales Demand Forecast in E-commerce Using a Long Short-Term Memory neural Network methodology*. International Conference on Neural Information Processing. https://doi.org/10.1007/978-3-030-36718-3_39

Pavlshenko, B. M. (2019, January 9). *Machine-Learning Models for Sales Time Series Forecasting. Data Mining and Processing*. https://doi.org/10.3390/data4010015

Bumpus, J. (2022, July 24). *Why you should reinstate the classic alarm clock.* CNN. Retrieved November 23, 2022, from https://www.cnn.com/style/article/why-you-should-reinstate-the-classic-alarm-clock-itlt/index.html

United Kingdom - Ecommerce. International Trade Administration | Trade.gov. (n.d.). Retrieved November 23, 2022, from https://www.trade.gov/country-commercial-guides/united-kingdom-ecommerce

Brownlee, J. (2016, July 21). *Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras*. Machine Learning Mastery. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

Appendix

ADS 506 Final

Project Code

John Chen

Azucena Faus

Setup

In [716…

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
import seaborn as sns
import dmba
from pathlib import Path
import datetime
from dateutil.parser import parse
import statsmodels.api as sm
#import statsmodels.formula.api as smf
from sklearn.metrics import accuracy_score
import plotly.express as px
import plotly.io as pio
from pandas import read_csv
import datetime
from numpy import log
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_predict
from statsmodels.tsa.stattools import acf
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.metrics import mean_squared_error
from dateutil.parser import parse
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt
from pandas.plotting import autocorrelation_plot
from pmdarima.arima import auto_arima
from statsmodels.tsa.seasonal import seasonal_decompose
from dateutil.parser import parse
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import r2_score, mean_absolute_error
from prettytable import PrettyTable
import warnings
warnings.filterwarnings('ignore')
warnings.simplefilter(action='ignore', category=FutureWarning)
```

Load/Observe Data

In [3]:

```
Retail_df = pd.read_csv('Online_Retail.csv')
Retail_df.sample(5)
```

Out[3]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
71850	542216	22983	CARD BILLBOARD FONT	12	1/26/2011 12:29	0.42	14911.0	EIRE

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
187279	552958	21174	POTTERING IN THE SHED METAL SIGN	12	5/12/2011 12:49	2.08	15498.0	United Kingdom
159144	550326	21212	PACK OF 72 RETROSPOT CAKE CASES	1	4/17/2011 13:05	0.55	14532.0	United Kingdom
249707	558906	82483	WOOD 2 DRAWER CABINET WHITE FINISH	2	7/4/2011 16:35	6.95	15555.0	United Kingdom
272433	560772	22720	SET OF 3 CAKE TINS PANTRY DESIGN	1	7/20/2011 16:12	10.79	NaN	United Kingdom

In [39]:
```
Retail_df.shape
```

Out[39]:
```
(541909, 8)
```

In [4]:
```
Retail_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   InvoiceNo    541909 non-null  object 
 1   StockCode    541909 non-null  object 
 2   Description  540455 non-null  object 
 3   Quantity     541909 non-null  int64  
 4   InvoiceDate  541909 non-null  object 
 5   UnitPrice    541909 non-null  float64
 6   CustomerID   406829 non-null  float64
 7   Country      541909 non-null  object 
dtypes: float64(2), int64(1), object(5)
memory usage: 33.1+ MB
```

In [6]:
```
Retail_df.describe()
```

Out[6]:

	Quantity	UnitPrice	CustomerID
count	541909.000000	541909.000000	406829.000000
mean	9.552250	4.611114	15287.690570
std	218.081158	96.759853	1713.600303
min	-80995.000000	-11062.060000	12346.000000
25%	1.000000	1.250000	13953.000000
50%	3.000000	2.080000	15152.000000
75%	10.000000	4.130000	16791.000000
max	80995.000000	38970.000000	18287.000000

Data Cleaning

Check for nulls

In [103…

```
Retail_noNA = Retail_df.dropna()
```

In [406…

```
pio.renderers.default='notebook'

dfg = Retail_noNA.groupby(['Description']).size().to_frame().sort_values([0],
                                                                         ascending = False
                                                                                reset_inde
dfg.columns = ['Description', 'count']
fig = px.histogram(dfg, x='Description', y = 'count',
                   title='Top Ten Item Descriptions Purchased on the Site')
fig.layout.yaxis.title.text = 'count'
fig.show()
```

missing values are not related to the forecast problem variables being used:

In [101…

```
Retail_df.isnull().values.any()
```

Out[101…

```
True
```

In [102…

```
Retail_df.isnull().sum()
```

Out[102…

```
InvoiceNo          0
StockCode          0
Description     1454
Quantity           0
InvoiceDate        0
UnitPrice          0
```

```
CustomerID     135080
Country             0
dtype: int64
```

Remove transactions that have to do with returns:

In [719…

```
# Remove transactions that were later returned with a negative
# quantity, so find the negative quantities, then
# the matching purchase for that return and remove both records
# from the data

# But first, make a copy of the dataframe to be modified:

Retail_df_NR = Retail_df.copy()

outlier_rows = Retail_df_NR['Quantity']< 0
outlier=Retail_df_NR[outlier_rows]
outlier

outlierI=outlier.copy()
# find records with negative of the negative quantity (positive)
outlierI['Quantity'] = -outlier['Quantity']

# Combine the data for invoice numbers for purchases that match
# Return invoices with negative quantity values:
commondf=pd.merge(Retail_df_NR,outlierI, on=['StockCode','CustomerID',
                                             'Quantity'])
commondf = commondf.rename(columns={'InvoiceNo_x': 'InvoiceNo'})

vector_invoices=commondf['InvoiceNo']
b=commondf.iloc[:,0].values
c=commondf.iloc[:,1].values

for i in range(0,len(b)):
    invoice_I = (Retail_df_NR['InvoiceNo'] == b[i]) &
                        (Retail_df_NR['StockCode'] == c[i])
    if Retail_df_NR[invoice_I].empty:
        print('')
    else:

        invoice_I2=Retail_df_NR[invoice_I].index
        Retail_df_NR=Retail_df_NR.drop(invoice_I2, axis=0)


#Retail_df_NR.shape
```

Matrix Generated that combines the original transaction with it's Return counterpart:

In [201…

```
# This was used to remove all the transactions
# that were actually refunded so as to not include false
# sales in our forecast.

# The matching item returns are found by matching CustomerID,
# Item StockCode, and Quantity inverted.

commondf.head()
```

Out[201…

	InvoiceNo	StockCode	Description_x	Quantity	InvoiceDate_x	UnitPrice_x	CustomerID	Country_x	InvoiceNo_y
0	536366	22632	HAND WARMER RED POLKA DOT	6	12/1/2010 8:28	1.85	17850.0	United Kingdom	C543611

	InvoiceNo	StockCode	Description_x	Quantity	InvoiceDate_x	UnitPrice_x	CustomerID	Country_x	InvoiceNo_y
1	536372	22632	HAND WARMER RED POLKA DOT	6	12/1/2010 9:01	1.85	17850.0	United Kingdom	C543611
2	536377	22632	HAND WARMER RED POLKA DOT	6	12/1/2010 9:34	1.85	17850.0	United Kingdom	C543611
3	536399	22632	HAND WARMER RED POLKA DOT	6	12/1/2010 10:52	1.85	17850.0	United Kingdom	C543611
4	536407	22632	HAND WARMER RED POLKA DOT	6	12/1/2010 11:34	1.85	17850.0	United Kingdom	C543611

In [199…
```
# new size of the retail data:
Retail_df_NR.shape
```

Out[199…
```
(532960, 8)
```

In [200…
```
# Since the for loop to get rid of
# returns lasted too long, saved the data
# for future reference on modeling etc:

Retail_df_NR.to_csv('Retail_NoReturn_Transactions.csv')
```

In [ ]:
```
p=sns.jointplot(x='vote_average',y='vote_count', data=Retail_df)
p.fig.suptitle("Relationship between Vote_Average and Vote_Count in Movie MetaData")
```

Remove Irrelevant and Unrealistic Records:

These have to do with fees, postage, and adjusted debt/credit transactions, not retail orders

In [202…
```
# Remove rows with negative quantities - these were used during adjustments
unreal_rows1 = Retail_df_NR[Retail_df_NR['Quantity'] <= 0].index


Retail_df_pre3 = Retail_df_pre2.drop(index=unreal_rows3)
unreal_rows4 = Retail_df_pre3[(Retail_df_pre3['StockCode'] == 'AMAZONFEE') |
                              (Retail_df_pre3['StockCode'] == 'DOT') |
                              (Retail_df_pre3['StockCode'] == 'M') |
                              (Retail_df_pre3['StockCode'] == 'B') |
                              (Retail_df_pre3['StockCode'] == 'POST')].index

Retail_df_pre4 = Retail_df_pre3.drop(index=unreal_rows4)
```

Data after removing both returns and irrelevant transactions that have nothing to do with sales:

In [203…
```
Retail_df_pre4.describe()
```

Out[203…

	Quantity	UnitPrice	CustomerID
count	519966.000000	519966.000000	391016.000000
mean	10.171529	3.235760	15300.029428

	Quantity	UnitPrice	CustomerID
std	36.451073	4.165506	1709.264898
min	1.000000	0.001000	12347.000000
25%	1.000000	1.250000	13971.000000
50%	3.000000	2.080000	15159.000000
75%	11.000000	4.130000	16800.000000
max	4800.000000	649.500000	18287.000000

In [204…
```
Retail_df_pre4.shape
```

Out[204…
```
(519966, 8)
```

In [102…
```
Retail_TimeSeries_df=Retail_df_pre4.copy()
```

In [244…
```
Retail_TimeSeries_df.isna().sum()
```

Out[244…
```
InvoiceNo            0
StockCode            0
Description          0
Quantity             0
InvoiceDate          0
UnitPrice            0
CustomerID      128950
Country              0
dtype: int64
```

Feature engineering SalesTotal:

In [245…
```
Retail_TimeSeries_df['Sales'] = (Retail_TimeSeries_df['Quantity'] *
                                 Retail_TimeSeries_df['UnitPrice'])
```

Exploratory Data Analysis

Observe time series and distributions based on size of orders

In [253…
```
rows_small_orders = (Retail_TimeSeries_df['Quantity'] < 100)
Retail_df_small_orders = Retail_TimeSeries_df[rows_small_orders]
Retail_df_small_orders.shape
```

Out[253…
```
(513663, 9)
```

In [221…
```
rows_medlarge_orders = (Retail_TimeSeries_df['Quantity'] >= 100) &
                                    (Retail_TimeSeries_df['Quantity'] < 1000)
Retail_df_medlarge_orders = Retail_TimeSeries_df[rows_medlarge_orders]
Retail_df_medlarge_orders.shape
```

Out[221…
```
(6201, 9)
```

In [222…
```
rows_large_orders = Retail_TimeSeries_df['Quantity'] >= 1000
```

```
Retail_df_large_orders = Retail_TimeSeries_df[rows_large_orders]
```

In [223…
```
Retail_df_large_orders.shape
```

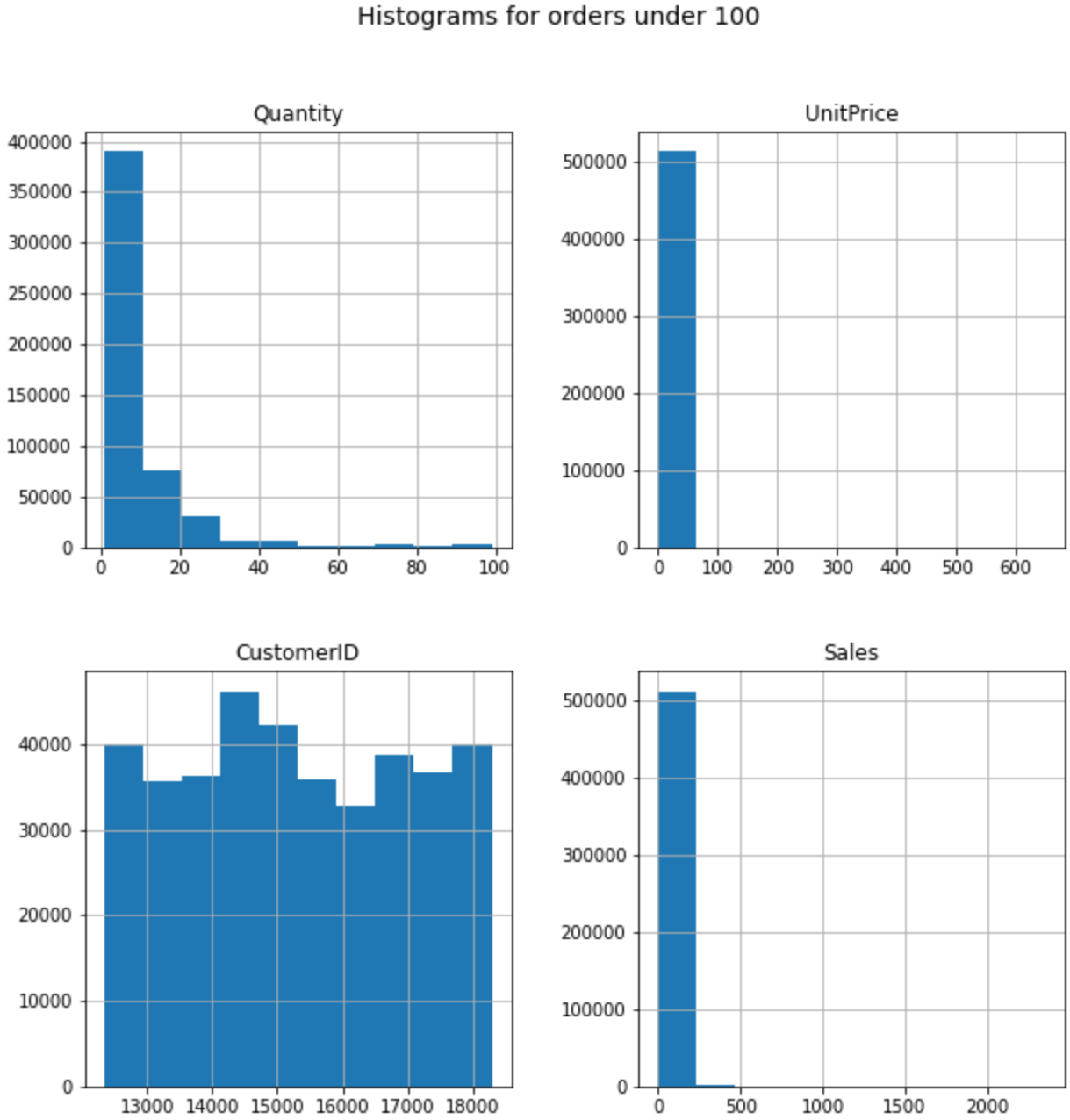
Out[223…
```
(102, 9)
```

Small order distributions (under 100 units)

In [405…
```
dfg_small_order = Retail_df_small_orders.groupby(['Description']).size().to_frame().
                                    sort_values([0], ascending = False).head(10).
                                                                    reset_index()
dfg_small_order.columns = ['Description', 'count']
fig = px.histogram(dfg_small_order, x='Description', y = 'count',
                   title='Top Ten Item Descriptions Purchased on Small Size orders')
fig.layout.yaxis.title.text = 'count'
fig.show()
```

In [256…
```
Retail_df_small_orders.hist(figsize=[10,10])
plt.suptitle("Histograms for orders under 100", fontsize=14)
```

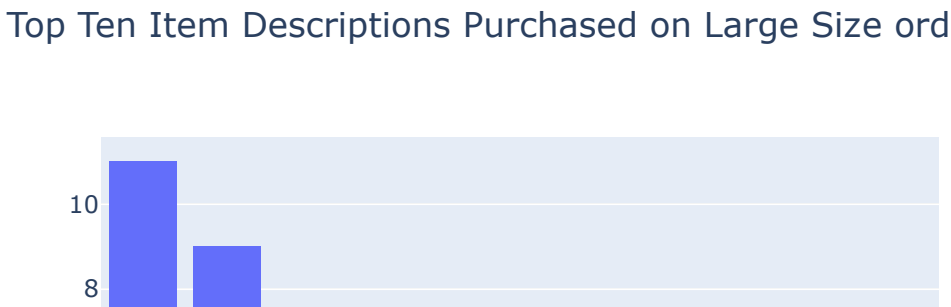
Out[256…
```
Text(0.5, 0.98, 'Histograms for orders under 100')
```

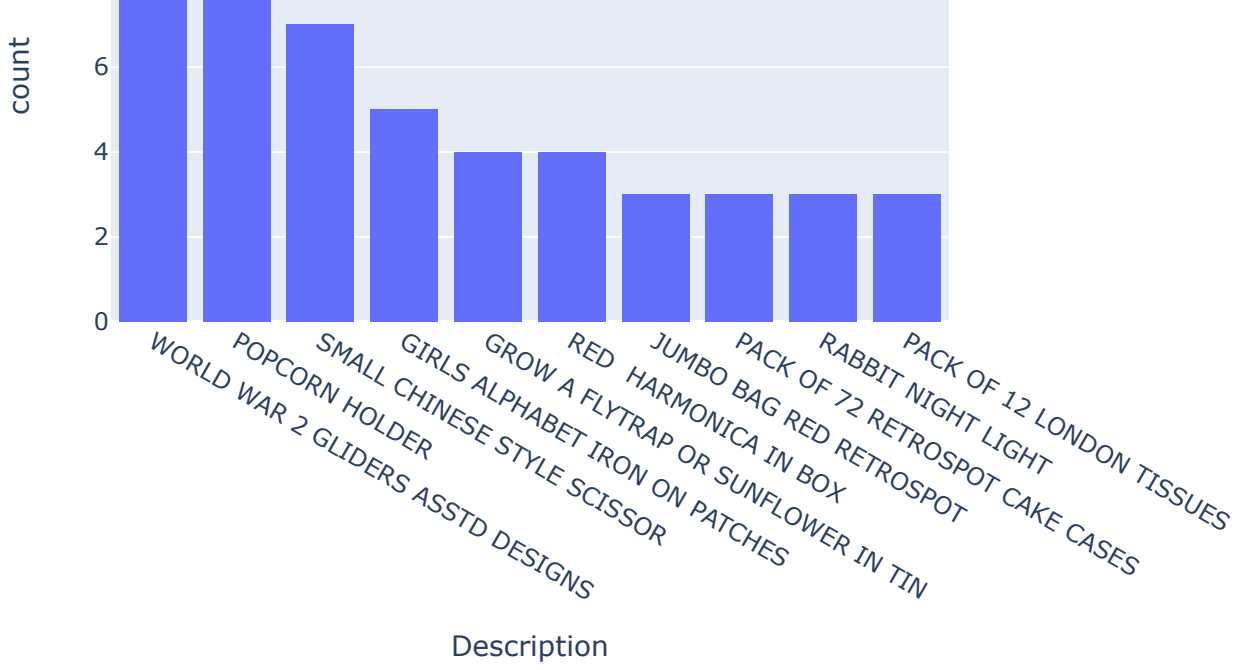


Large (>1000) order distributions:

In [407…

```
dfg_large_order = Retail_df_large_orders.groupby(['Description']).size().to_frame().
                                                                    sort_values([0],
                                                                    ascending = False).head(10
                                                                    reset_index()
dfg_large_order.columns = ['Description', 'count']
fig = px.histogram(dfg_large_order, x='Description', y = 'count',
                   title='Top Ten Item Descriptions Purchased on Large Size orders')
fig.layout.yaxis.title.text = 'count'
fig.show()
```



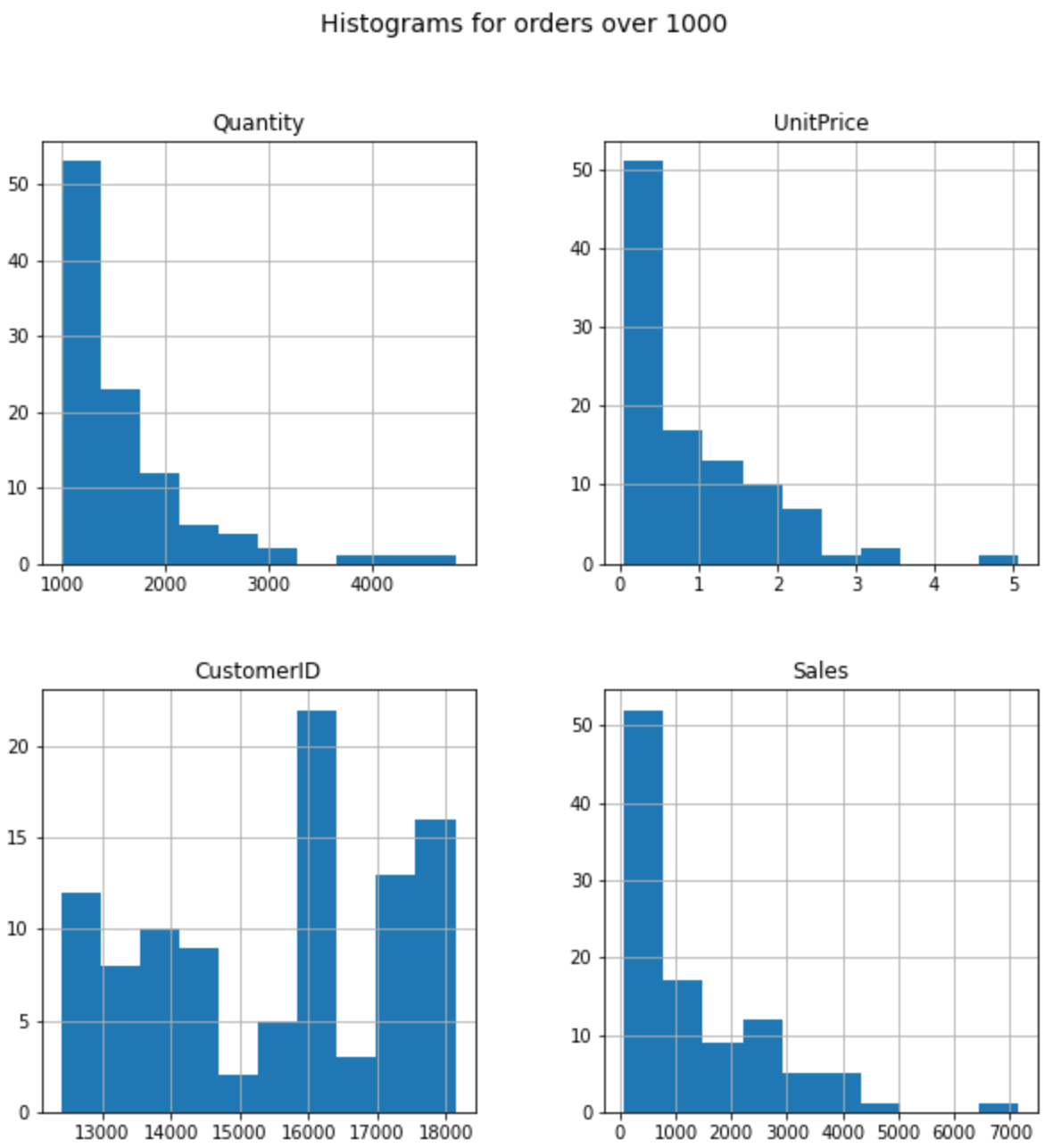


In [224…

```
Retail_df_large_orders.hist(figsize=[10,10])
plt.suptitle("Histograms for orders over 1000", fontsize=14)
```

Out[224…

```
Text(0.5, 0.98, 'Histograms for orders over 1000')
```



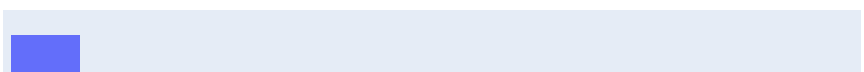
In [225…

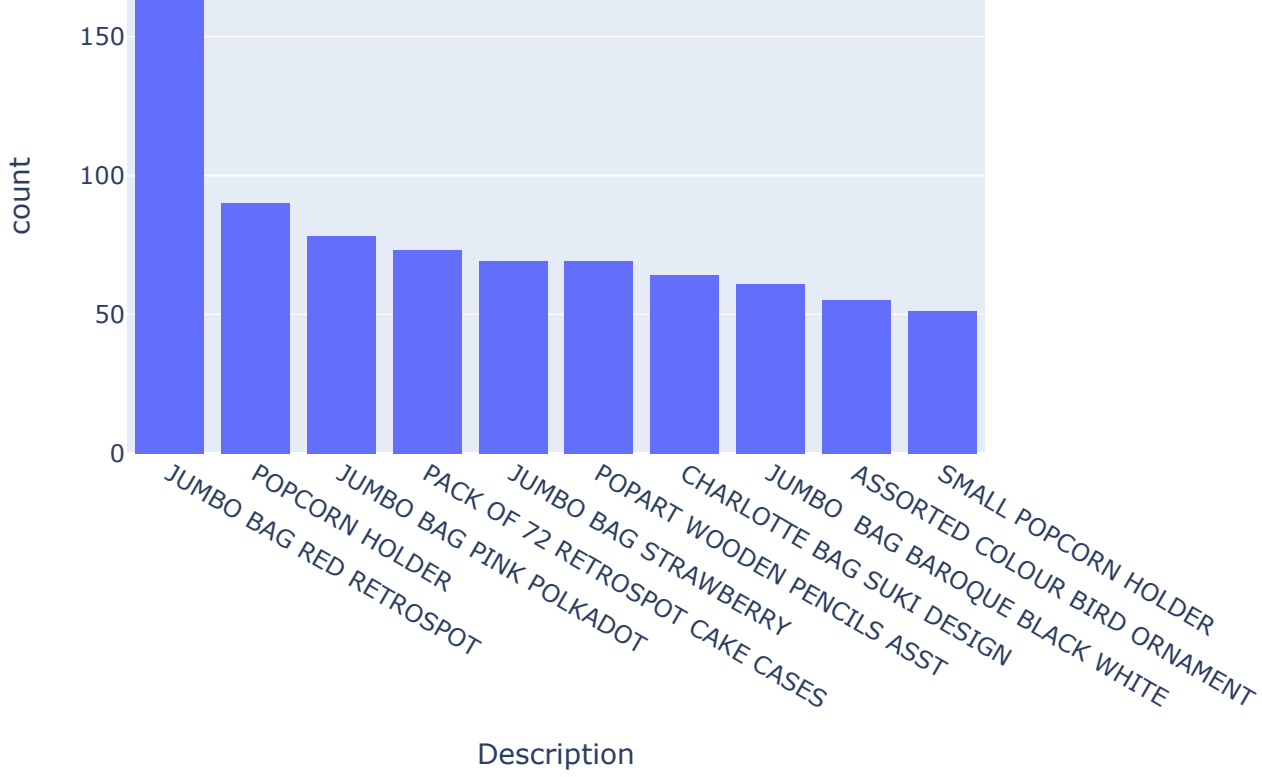
```
Retail_df_large_orders.to_csv('Retail_df_large_orders.csv')
```

Medium (100-1000 units) order Distributions:

In [408…

```
dfg_medium_order = Retail_df_medlarge_orders.groupby(['Description']).size().to_frame().
                                                    sort_values([0],
                                                                ascending = False).head(10).
                                                                reset_index()
dfg_medium_order.columns = ['Description', 'count']
fig = px.histogram(dfg_medium_order, x='Description', y = 'count',
                   title='Top Ten Item Descriptions Purchased on Medium Size orders')
fig.layout.yaxis.title.text = 'count'
fig.show()
```



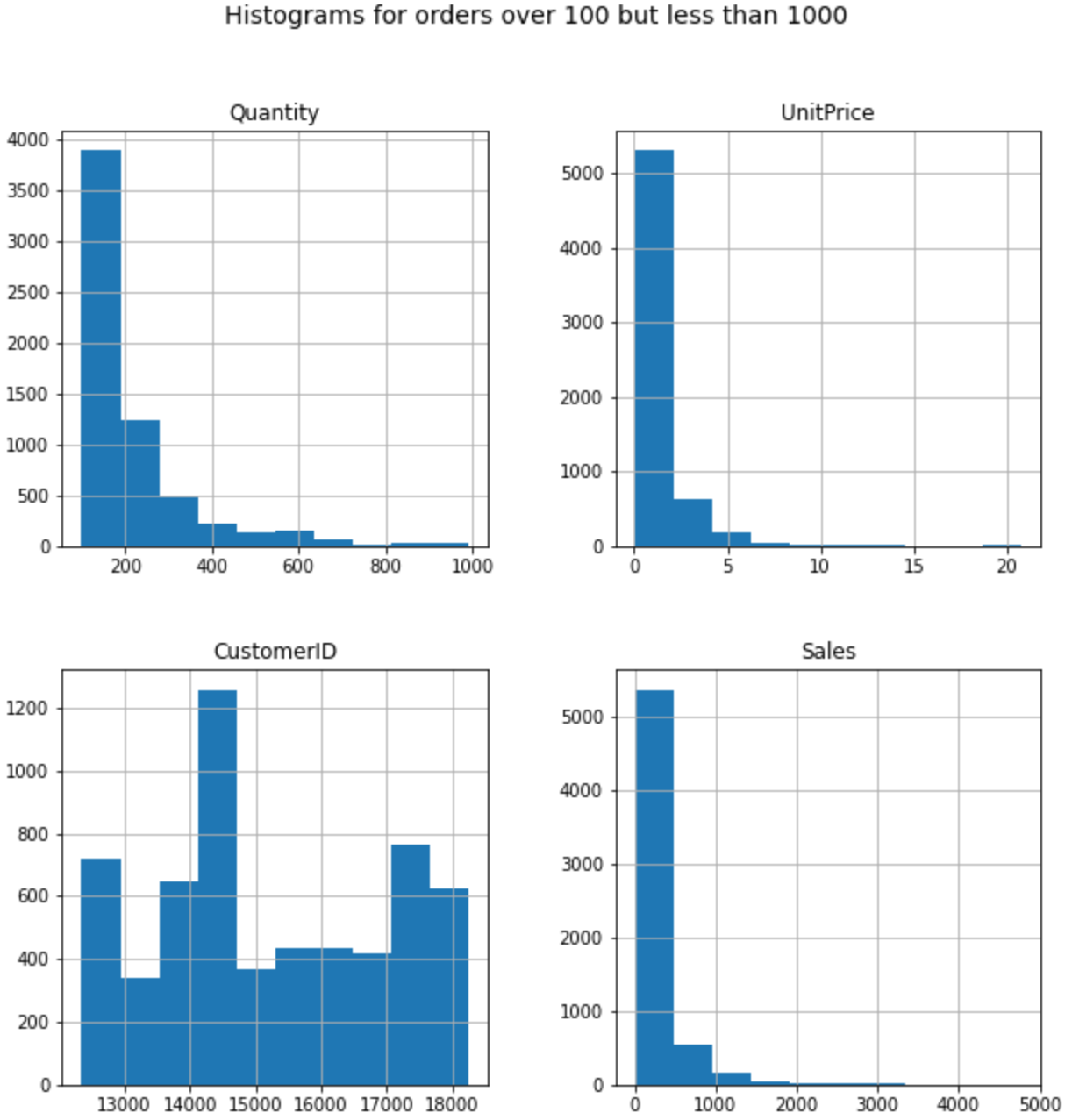


In [226…

```
Retail_df_medlarge_orders.hist(figsize=[10,10])
plt.suptitle("Histograms for orders over 100 but less than 1000", fontsize=14)
```

Out[226…

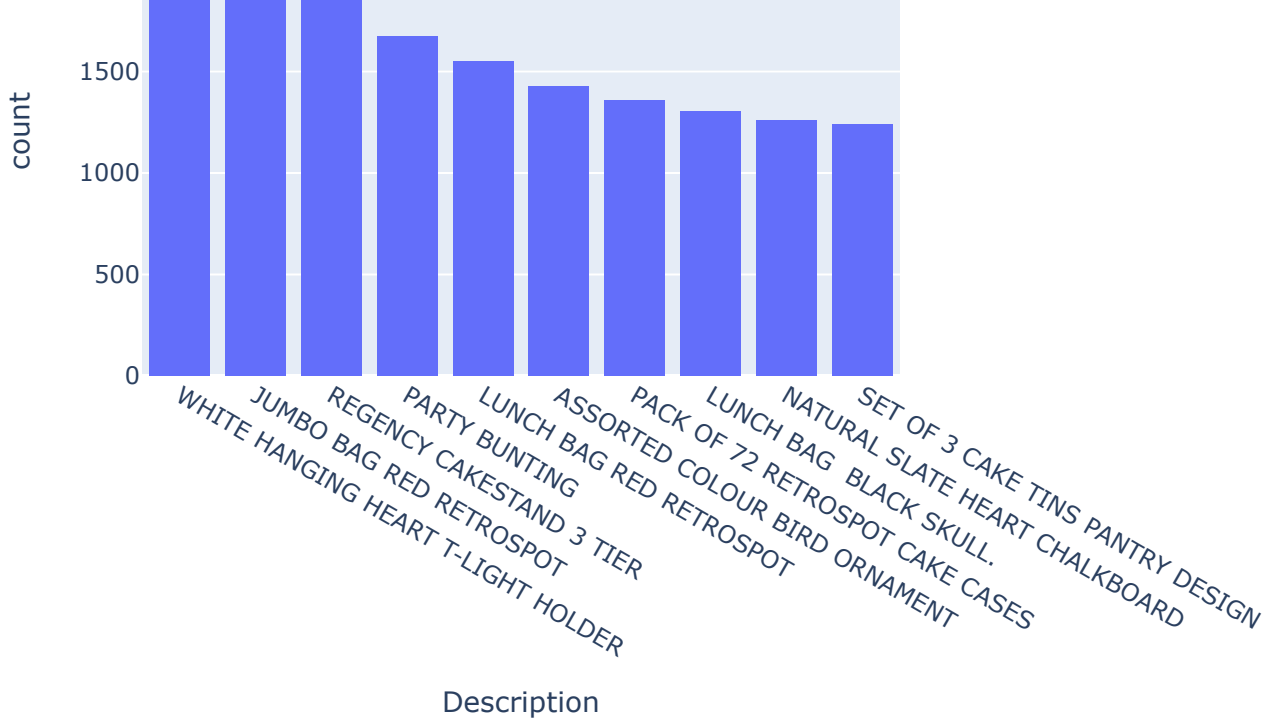
```
Text(0.5, 0.98, 'Histograms for orders over 100 but less than 1000')
```



Most popular items purchased:

In [409…

```
pio.renderers.default='notebook'

dfg = Retail_TimeSeries_df.groupby(['Description']).size().
                                                    to_frame().sort_values([0],
                                                    ascending = False).head(10).reset_index()
dfg.columns = ['Description', 'count']
fig = px.histogram(dfg, x='Description', y = 'count',
                   title='Top Ten Item Descriptions Purchased on the Site')
fig.layout.yaxis.title.text = 'count'
fig.show()
```



In [233…
```
MostCommonItem = Retail_TimeSeries_df[Retail_TimeSeries_df['Description'].
                                      str.contains('WHITE HANGING HEART T-LIGHT HOLDER',
                                                   na=False)]
```

In [234…
```
MostCommonItem.shape
```

Out[234…
```
(2293, 9)
```

In [236…
```
MostCommonItem.describe()
```

Out[236…

	Quantity	UnitPrice	CustomerID	Sales
count	2293.000000	2293.000000	1998.000000	2293.000000
mean	14.501526	3.221029	15558.954454	41.273973
std	43.009513	0.995402	1618.141817	129.383443
min	1.000000	2.550000	12370.000000	2.950000
25%	3.000000	2.950000	14221.000000	8.850000
50%	6.000000	2.950000	15584.000000	17.700000
75%	12.000000	2.950000	16931.000000	35.400000
max	1010.000000	6.770000	18283.000000	3272.400000

In [238…
```
MostCommonItem.Sales.sum()
```

Out[238…
```
94641.22
```

Particular item order Distributions: Clocks

In [227…
```
clock = Retail_TimeSeries_df[Retail_TimeSeries_df['Description'].
                             str.contains('CLOCK',
```

```
                                                na=False)]
```

In [441…
```
clock.shape
```

Out[441…
```
(7025, 9)
```

In [259…
```
clock_test = Retail_df_small_orders[Retail_df_small_orders['Description'].
                                    str.contains('CLOCK',
                                                 na=False)]
clock_test.shape
```

Out[259…
```
(6990, 9)
```

Most clock sales come from small orders with some coming from medium sized orders

In [219…
```
clock.shape
```

Out[219…
```
(7025, 8)
```

In [237…
```
clock.describe()
```

Out[237…

	Quantity	UnitPrice	CustomerID	Sales
count	7025.000000	7025.000000	5726.000000	7025.000000
mean	5.303203	6.096231	15250.322040	23.958272
std	18.973171	3.737963	1739.507654	76.379316
min	1.000000	0.190000	12347.000000	0.190000
25%	1.000000	3.750000	13767.000000	8.290000
50%	2.000000	3.750000	15178.000000	15.000000
75%	4.000000	8.500000	16729.000000	19.900000
max	620.000000	49.960000	18280.000000	2662.200000

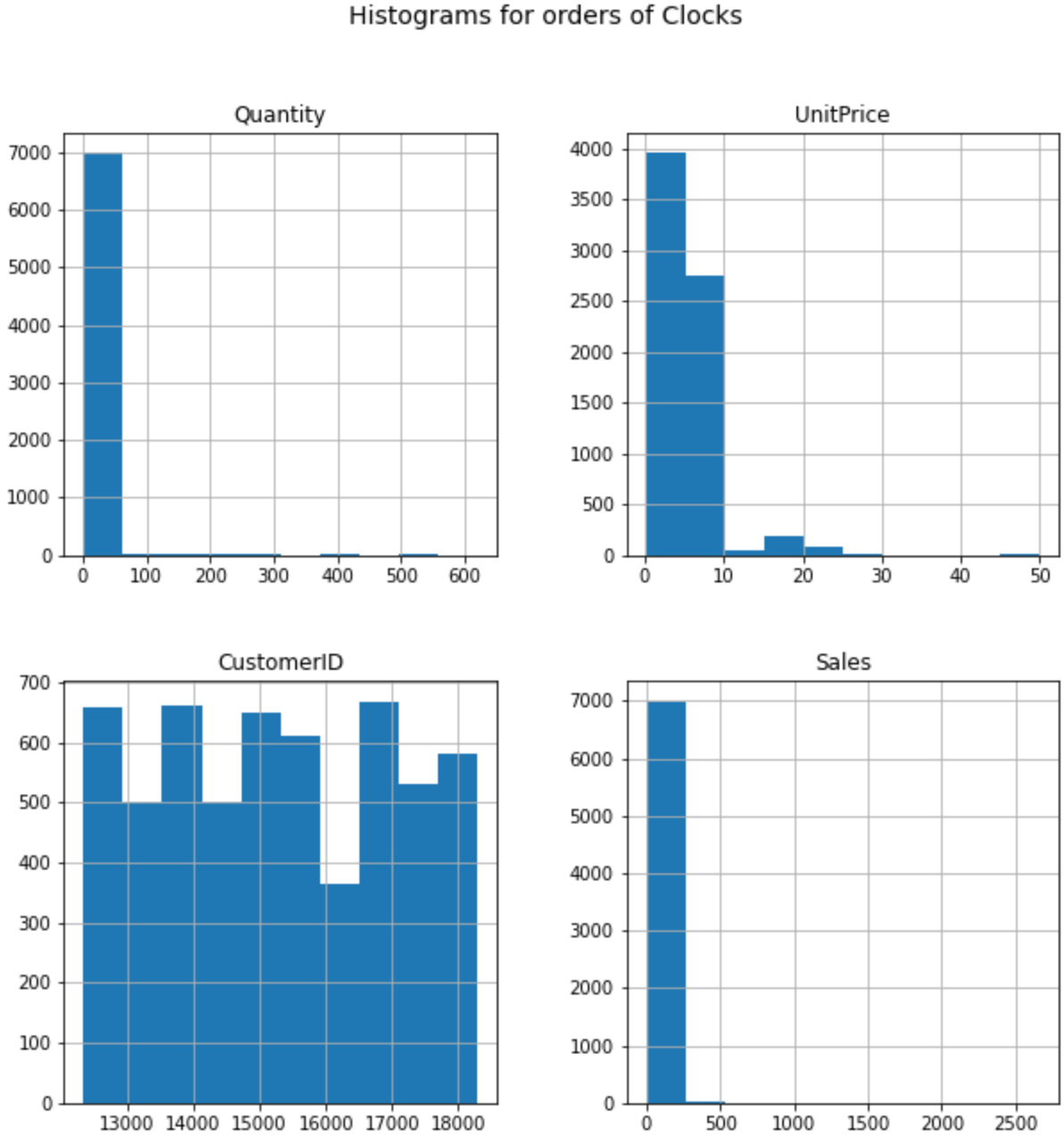
In [239…
```
clock.Sales.sum()
```

Out[239…
```
168306.86000000002
```

Sales from clocks alone amount to a total of 168,306 as opposed to the most popular item that amounts to 94,641 and with projected interest in clocks going up, must look at initial forecasts for this item.

In [228…
```
clock.hist(figsize=[10,10])
plt.suptitle("Histograms for orders of Clocks", fontsize=14)
```

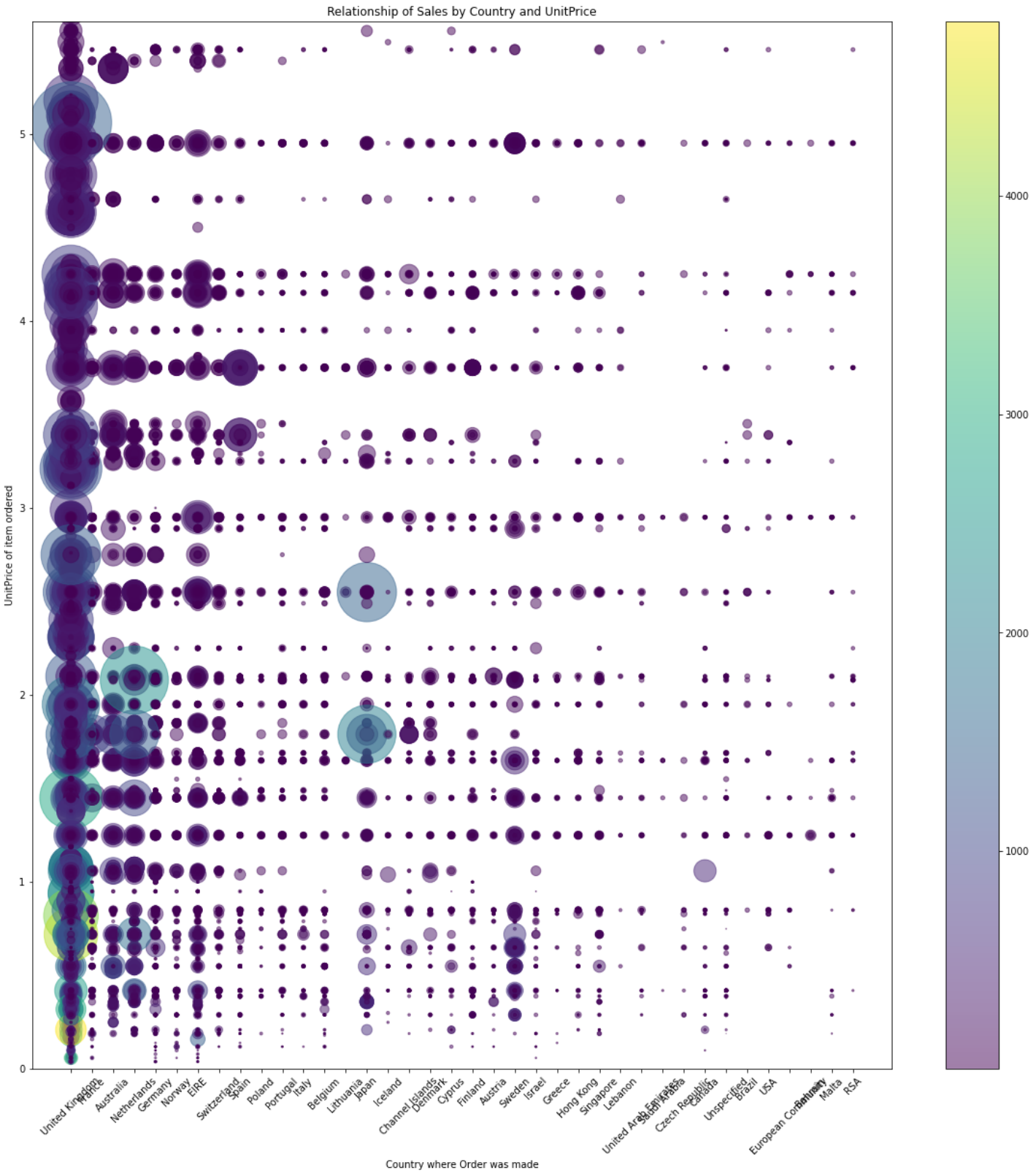
Out[228…
```
Text(0.5, 0.98, 'Histograms for orders of Clocks')
```



In [ ]:
```
clock['Clock_Purchase'] = 1
```

Relationship between Sales and other variables

In [264…
```
plt.figure(figsize=(20,20))
plt.scatter(Retail_TimeSeries_df['Country'], Retail_TimeSeries_df['UnitPrice'],
            s=Retail_TimeSeries_df['Sales'], c=Retail_TimeSeries_df['Quantity'],
            alpha=0.5
           )
plt.gca().update(dict(title='Relationship of Sales by Country and UnitPrice',
                      xlabel='Country where Order was made', ylabel='UnitPrice of item orc
                      ylim=(0,5.6)))
plt.xticks(rotation=45)
plt.colorbar()
plt.show()
```

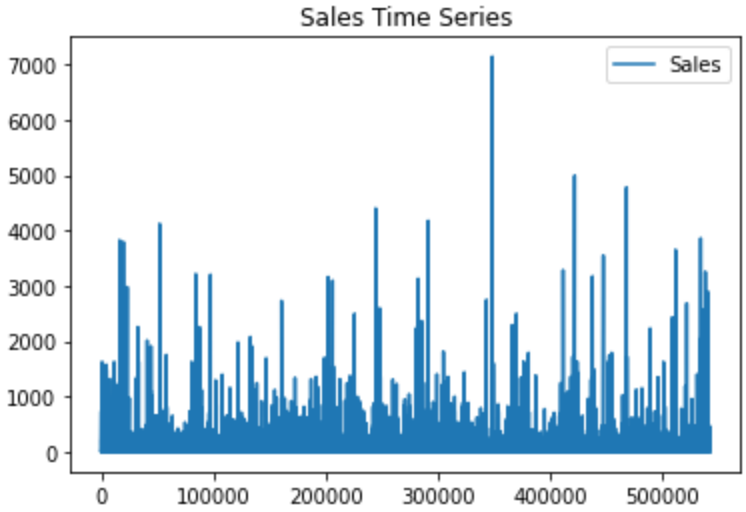


Most orders are from the UK and most are small size orders (darker purple). Large Sale values span the unit price range.

Time Series Plots

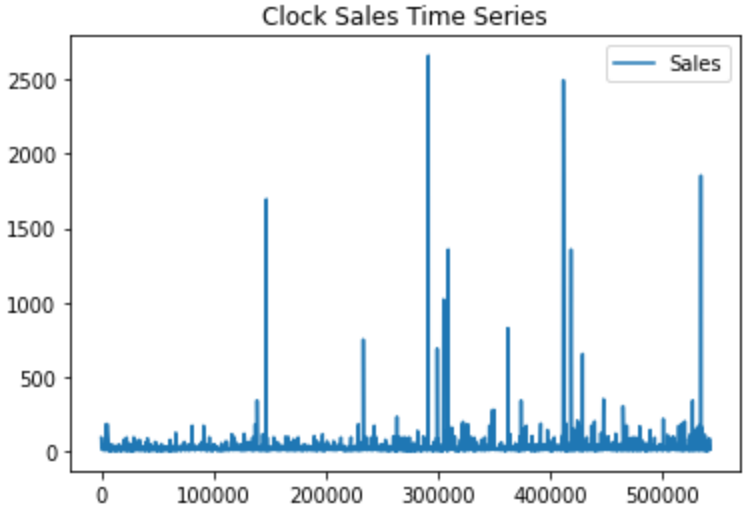
In [247…

```
Retail_SalesOnly = Retail_TimeSeries_df.copy()
Retail_SalesOnly.drop(columns=['InvoiceNo', 'CustomerID','UnitPrice','Quantity'],
                      inplace=True)

Retail_Sales.plot()
plt.title('Sales Time Series')
plt.show()
```



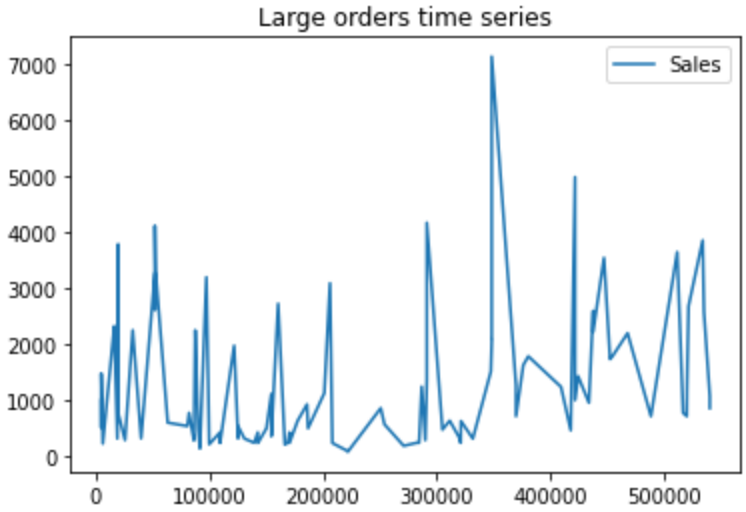
In [248…

```
Clock_SalesOnly = clock.copy()
Clock_SalesOnly.drop(columns=['InvoiceNo', 'CustomerID','UnitPrice','Quantity'],
                     inplace=True)

Clock_SalesOnly.plot()
plt.title('Clock Sales Time Series')
plt.show()
```



In [249…

```
series_time_retail = read_csv('Retail_df_large_orders.csv', header=0, index_col=0,
                              parse_dates=True,
                              squeeze=True)
series_time_retail.drop(columns=['InvoiceNo', 'CustomerID','UnitPrice','Quantity'],
                        inplace=True)

series_time_retail.plot()
plt.title('Large orders time series')
plt.show()
```

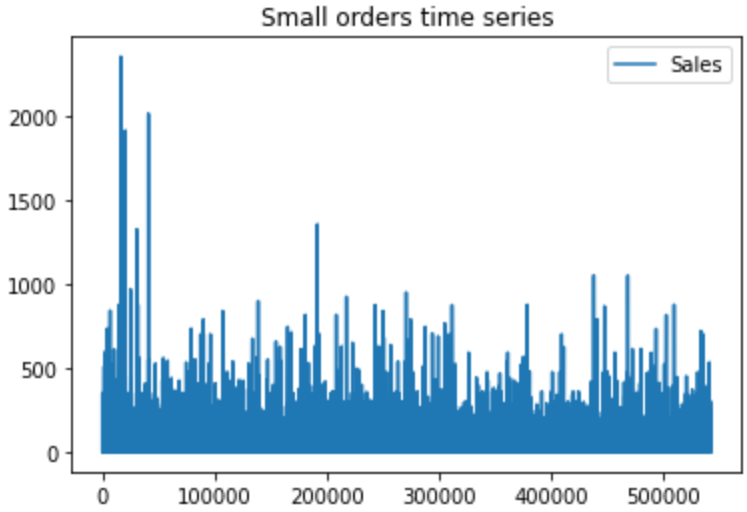


In [250…

```
series_time_retail2 = Retail_df_medlarge_orders.copy()
series_time_retail2.drop(columns=['InvoiceNo', 'CustomerID','UnitPrice','Quantity'],
                         inplace=True)

series_time_retail2.plot()
plt.title('Medium orders time series')
plt.show()
```

In [260…

```
series_time_retail3 = Retail_df_small_orders.copy()
series_time_retail3.drop(columns=['InvoiceNo', 'CustomerID','UnitPrice','Quantity'],
                         inplace=True)

series_time_retail3.plot()
plt.title('Small orders time series')
plt.show()
```



- Most orders originate from the UK, which is where the company is based
- Most sales are small quantities (purple color)

Clock Modeling Pre-Processing Sales Data Set:

Time series by Date:

In [410…
```
clock.to_csv('Clock_Retail_UKdata.csv')
```

In [102…
```
Clock_TimeSeries_date_df = read_csv('Clock_Retail_UKdata.csv', header=0, index_col=0,
                                    parse_dates=True, squeeze=True)
Clock_TimeSeries_date_df['InvoiceDate'] = pd.to_datetime(Clock_TimeSeries_date_df['Invoice

Clock_TimeSeries_date_df=Clock_TimeSeries_date_df.set_index('InvoiceDate')
```

In [102…
```
Clock_TimeSeries_date_df.head()
```

Out[102…

InvoiceDate	InvoiceNo	StockCode	Description	Quantity	UnitPrice	CustomerID	Country	Sales
2010-12-01 08:45:00	536370	22728	ALARM CLOCK BAKELIKE PINK	24	3.75	12583.0	France	90.0
2010-12-01 08:45:00	536370	22727	ALARM CLOCK BAKELIKE RED	24	3.75	12583.0	France	90.0
2010-12-01 08:45:00	536370	22726	ALARM CLOCK BAKELIKE GREEN	12	3.75	12583.0	France	45.0
2010-12-01 09:45:00	536382	22726	ALARM CLOCK BAKELIKE GREEN	4	3.75	16098.0	United Kingdom	15.0
2010-12-01 10:03:00	536389	22193	RED DINER WALL CLOCK	2	8.50	12431.0	Australia	17.0

Focus only on UK sales:

In [102…
```
Clock_TimeSeries_date_df =
            Clock_TimeSeries_date_df[Clock_TimeSeries_date_df['Country'].
```

```
                                    str.contains('United Kingdom',
                                                 na=False)]
```

In [102…

```
Clock_TimeSeries_date_df.head()
```

Out[102…

InvoiceDate	InvoiceNo	StockCode	Description	Quantity	UnitPrice	CustomerID	Country	Sales
2010-12-01 09:45:00	536382	22726	ALARM CLOCK BAKELIKE GREEN	4	3.75	16098.0	United Kingdom	15.0
2010-12-01 10:47:00	536395	22730	ALARM CLOCK BAKELIKE IVORY	4	3.75	13767.0	United Kingdom	15.0
2010-12-01 10:47:00	536395	22727	ALARM CLOCK BAKELIKE RED	8	3.75	13767.0	United Kingdom	30.0
2010-12-01 10:47:00	536395	22729	ALARM CLOCK BAKELIKE ORANGE	8	3.75	13767.0	United Kingdom	30.0
2010-12-01 10:47:00	536395	22726	ALARM CLOCK BAKELIKE GREEN	8	3.75	13767.0	United Kingdom	30.0

In [102…

```
Clock_TimeSeries_date_df.shape
```

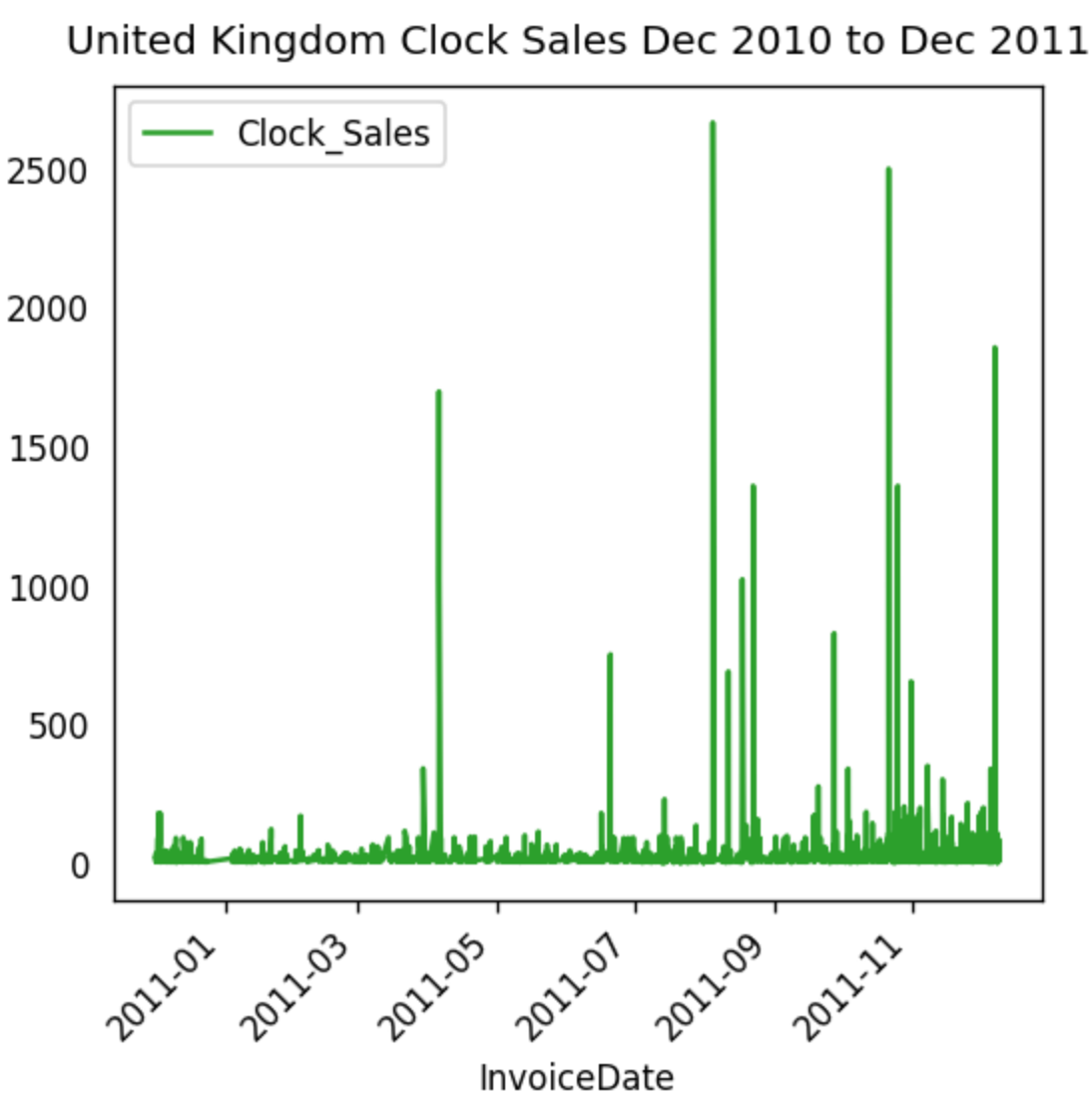
Out[102…

```
(6281, 8)
```

In [127…

```
UK_clock_ts = Clock_TimeSeries_date_df.copy()
UK_clock_ts.drop(columns=['InvoiceNo', 'CustomerID','UnitPrice','Quantity',
                          'StockCode','Description','Country'],inplace=True)
plt.figure(figsize=(5,3))
UK_clock_ts.plot(color='tab:green')
plt.title('United Kingdom Clock Sales Dec 2010 to Dec 2011')
plt.xticks(rotation=45)
plt.legend(['Clock_Sales'])
plt.show()
```

```
<Figure size 600x360 with 0 Axes>
```



Sales per Day:

In [992…
```
UK_clock_ts.head()
```

Out[992…

InvoiceDate	Sales
2010-12-01 09:45:00	15.0
2010-12-01 10:47:00	15.0
2010-12-01 10:47:00	30.0
2010-12-01 10:47:00	30.0
2010-12-01 10:47:00	30.0

In [993…
```
UK_clock_ts.shape
```

Out[993…
```
(6269, 1)
```

Clock Dataset with only Daily Sales and Date Index:

In [102…
```
UK_DailyClock_ts = UK_clock_ts.iloc[:,0].resample('d').sum()
```

In [103…
```
UK_DailyClock_df=pd.DataFrame(UK_DailyClock_ts)
```

In [103…
```
UK_DailyClock_df.head()
```

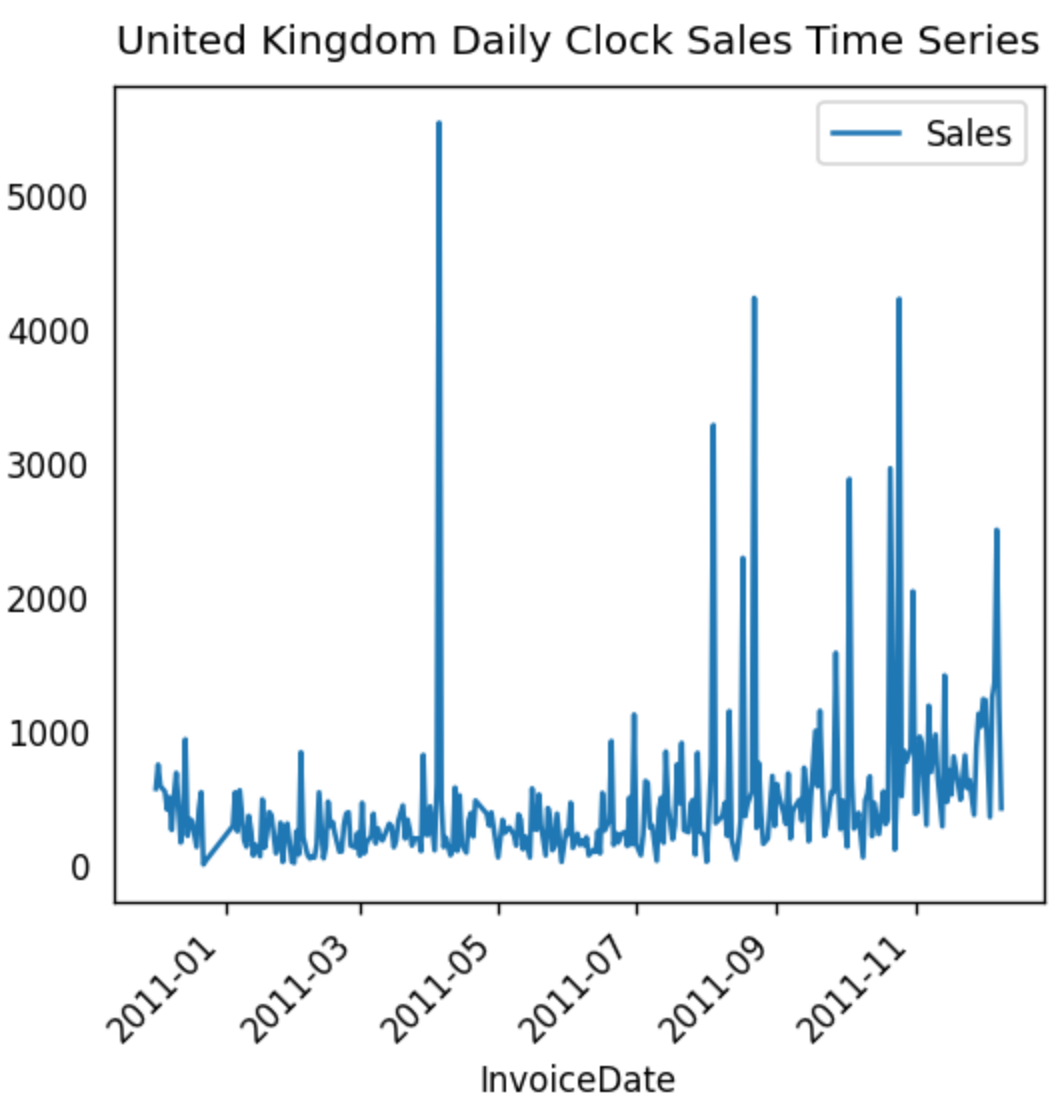
Out[103…

InvoiceDate	Sales
2010-12-01	568.40
2010-12-02	747.25
2010-12-03	587.62
2010-12-04	0.00
2010-12-05	547.25

In [103…
```
UK_DailyClock_df = UK_DailyClock_df[UK_DailyClock_df['Sales'] > 1]
```

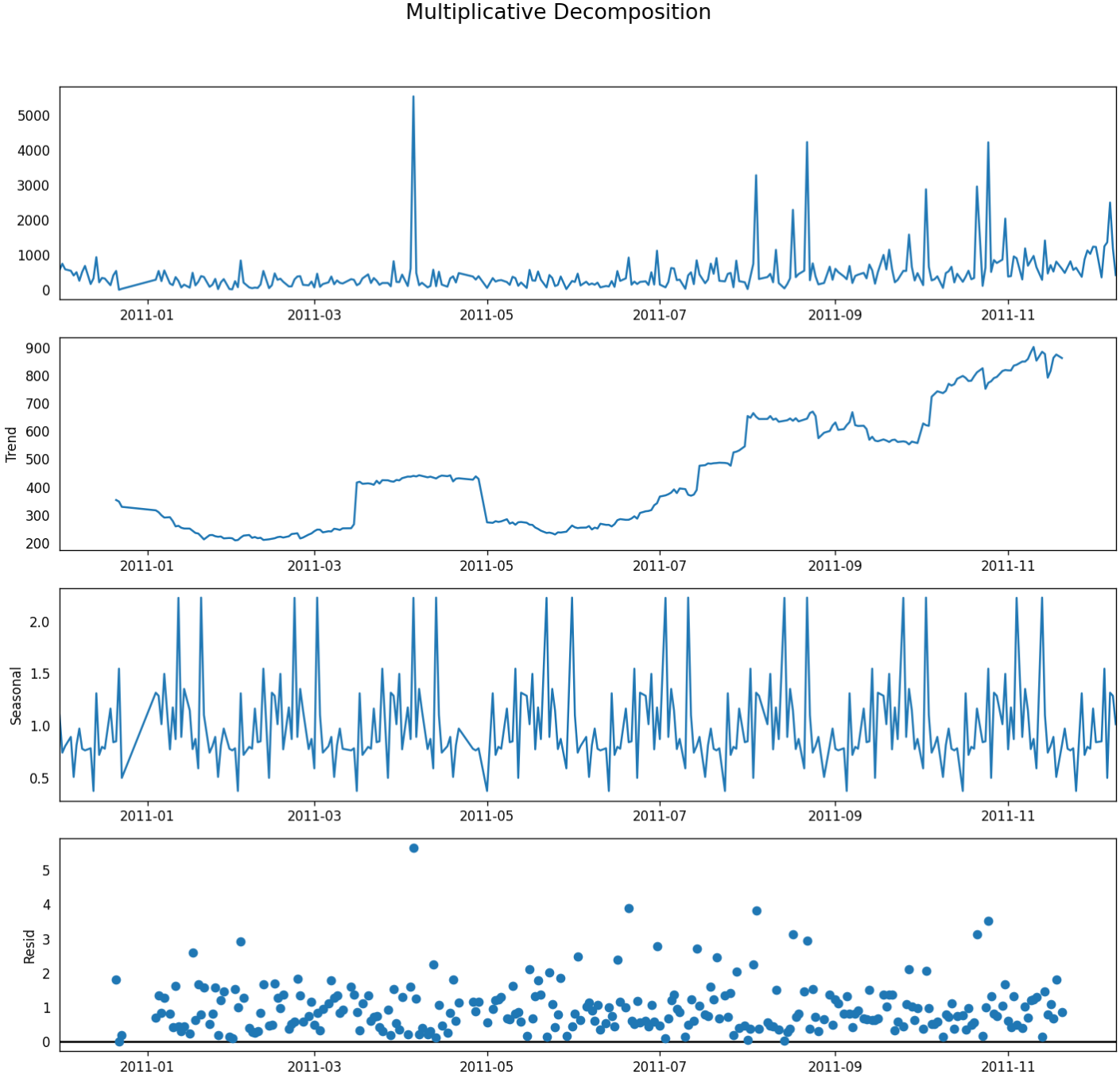
In [103…
```
plt.figure(figsize=(6,4))
UK_DailyClock_df.plot()

plt.title('United Kingdom Daily Clock Sales Time Series')
plt.xticks(rotation=45)
plt.show()
```

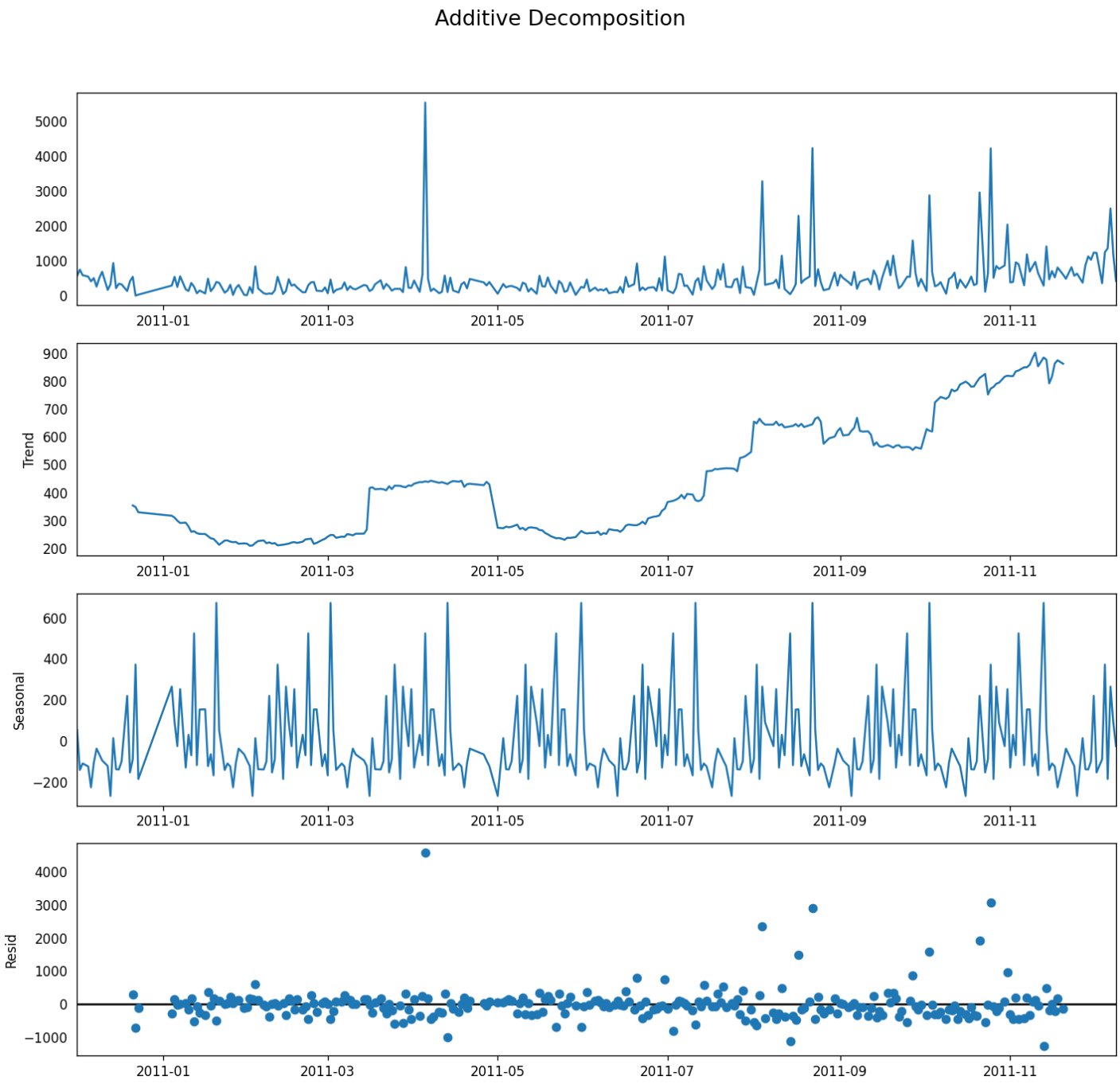
```
<Figure size 720x480 with 0 Axes>
```



In [103…

```
# Decomposition
# Decomposition of a time series can be performed by considering
# the series as an additive or multiplicative combination of the
# base level, trend, seasonal index and the residual term.

# Multiplicative Decomposition
multiplicative_decomposition = seasonal_decompose(UK_DailyClock_df,
                                                  model='multiplicative',
                                                  period=35)

# Additive Decomposition
additive_decomposition = seasonal_decompose(UK_DailyClock_df,
                                            model='additive',
                                            period=35)

# Plot
plt.rcParams.update({'figure.figsize': (12,12)})
multiplicative_decomposition.plot().suptitle('Multiplicative Decomposition',
                                             fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

additive_decomposition.plot().suptitle('Additive Decomposition',
                                       fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

plt.show()
```


Multiplicative Decomposition
Trend
Seasonal
Resid
2011-01
2011-03
2011-05
2011-07
2011-09
2011-11



If we look at the residuals of the additive decomposition closely, it has some pattern left over.

The multiplicative decomposition, looks quite random which is good. So ideally, multiplicative decomposition should be preferred for this particular series.

Stationarity and Correlation Tests on Time Series:

In [103…

```
# Check for stationarity and if the signal is a random walk:

result = adfuller(UK_DailyClock_df.dropna())
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
```

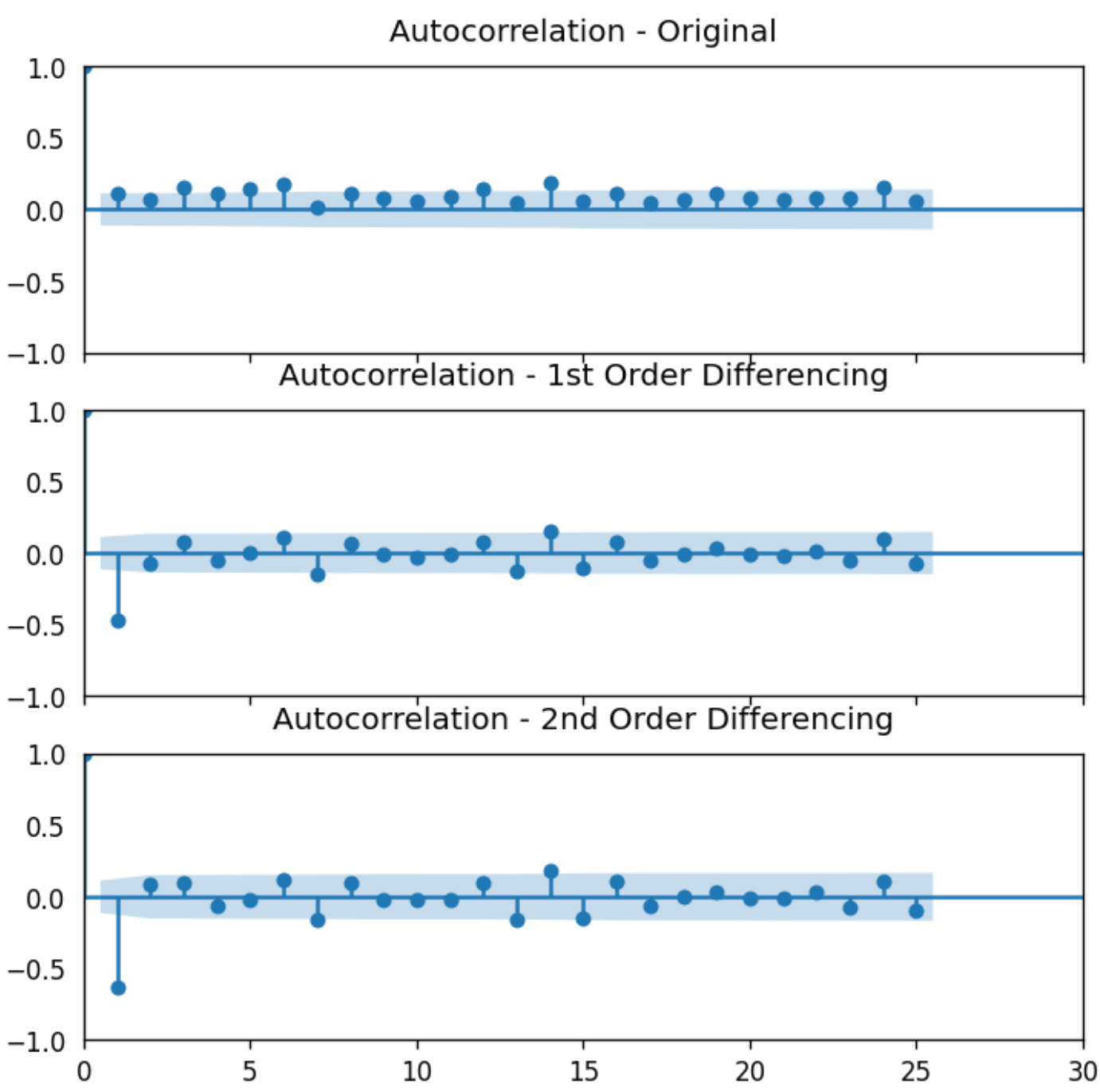
```
ADF Statistic: -4.112281
p-value: 0.000924
```

- Null Hypothesis: series is non-stationary

- Alternate Hypothesis: series is stationary
    - p-value is < 0.05 so we can reject the null hypothesis.
    - Therefore, the series is stationary

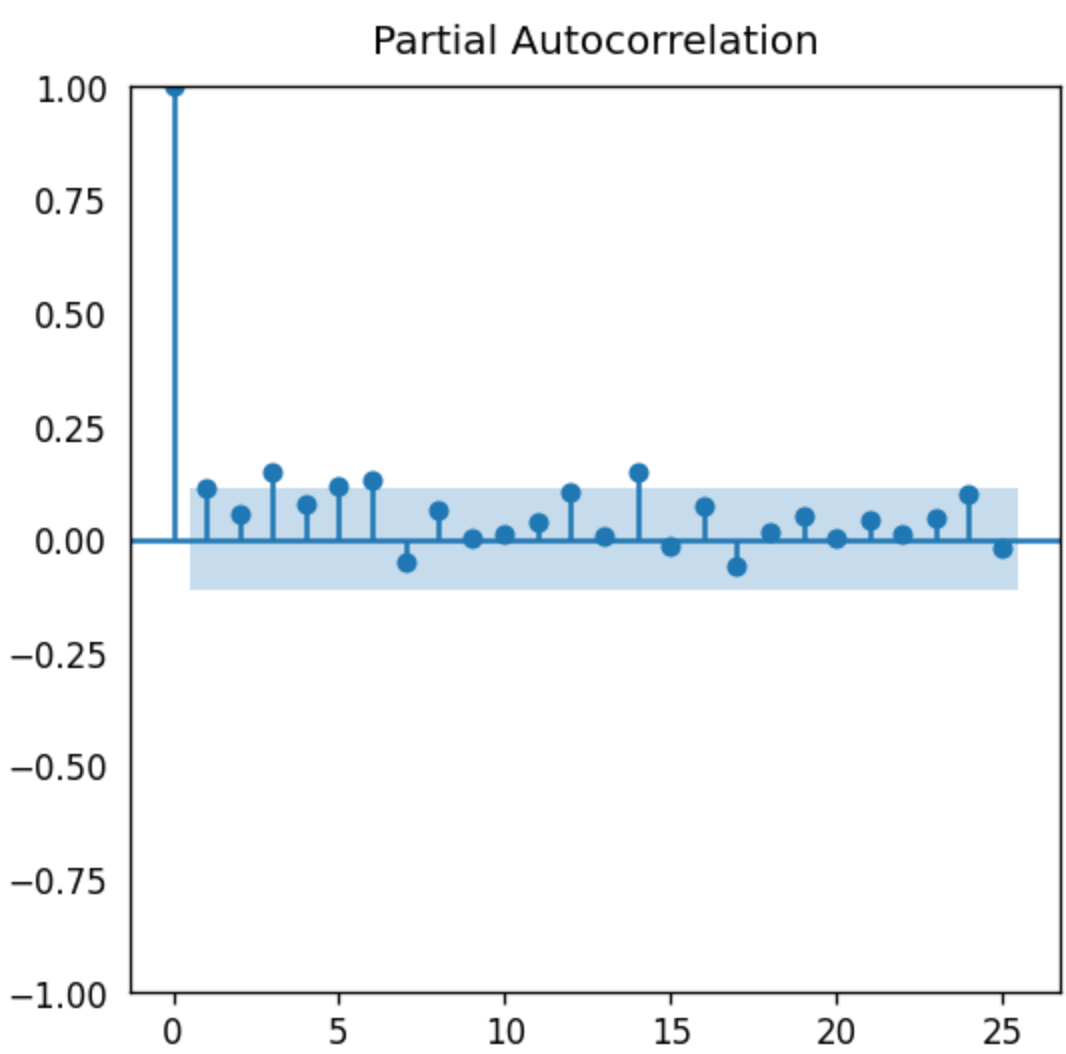
In [103…

```
plt.rcParams.update({'figure.figsize':(7,7), 'figure.dpi':120})
# Import data

# Original Series
fig, axes = plt.subplots(3, sharex=True)

plot_acf(UK_DailyClock_df, ax=axes[0])
axes[0].set_title('Autocorrelation - Original')

plot_acf(UK_DailyClock_df.diff().dropna(), ax=axes[1])
axes[1].set_title('Autocorrelation - 1st Order Differencing')

plot_acf(UK_DailyClock_df.diff().diff().dropna(), ax=axes[2])
axes[2].set_title('Autocorrelation - 2nd Order Differencing')

plt.xlim([0, 30])
plt.show()
```



In [104…

```
# PACF plot
plt.rcParams.update({'figure.figsize':(5,5), 'figure.dpi':120})

pacf = plot_pacf(UK_DailyClock_df['Sales'], lags=25)
plt.title('Partial Autocorrelation')
plt.show()
```

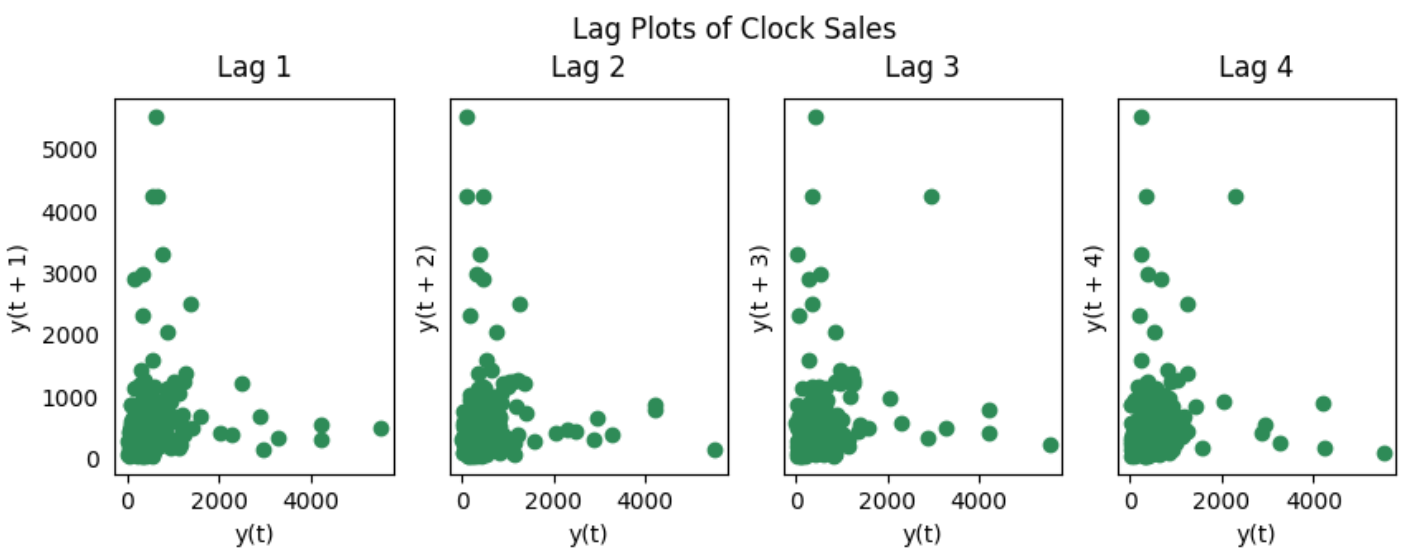


The above plot can be used to determine the order of AR model. You may note that a correlation value up to order 3 is high enough. Thus, we will train the AR model of order 3.

If partial autocorrelation values are close to 0, then values between observations and lagged observations are not correlated with one another. Inversely, partial autocorrelations with values close to 1 or -1 indicate that there exists strong positive or negative correlations between the lagged observations of the time series.

In [104…

```
# Lag Plots

from pandas.plotting import lag_plot
plt.rcParams.update({'ytick.left' : False, 'axes.titlepad':10})

# Plot
fig, axes = plt.subplots(1, 4, figsize=(10,3), sharex=True,
                         sharey=True, dpi=100)
for i, ax in enumerate(axes.flatten()[:4]):
    lag_plot(UK_DailyClock_df['Sales'], lag=i+1, ax=ax, c='seagreen')
    ax.set_title('Lag ' + str(i+1))

fig.suptitle('Lag Plots of Clock Sales', y=1.05)
plt.show()
```



A Lag plot is a scatter plot of a time series against a lag of itself. It is normally used to check for autocorrelation. If there is any pattern existing in the series, the series is autocorrelated. If there is no such pattern, the series is likely to be random white noise.

Naive Forecast Method

30 day prediction

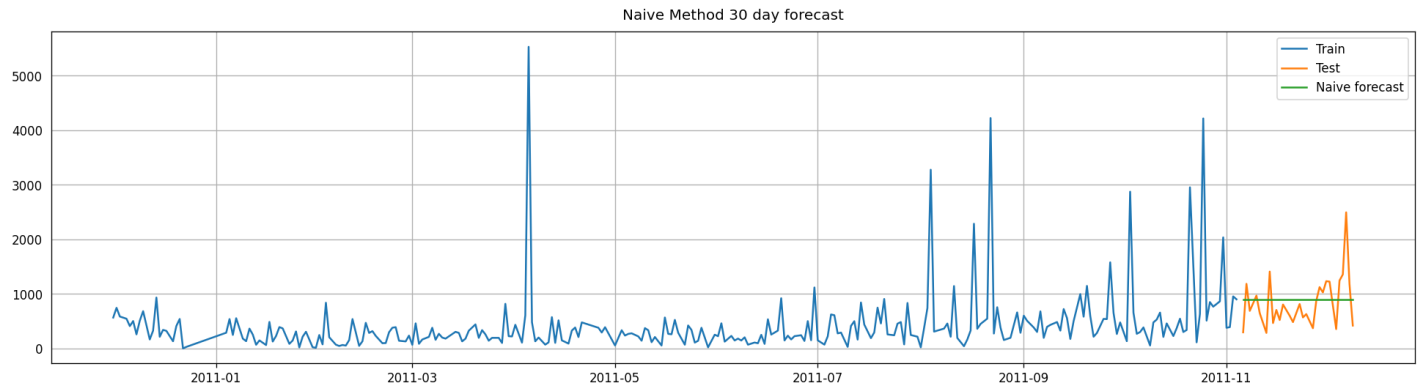
In [116…

```
# Split Train / Test

train_length = 273
train = UK_DailyClock_df[0:train_length]
test = UK_DailyClock_df[train_length:]
print(len(train))
print('')
print(len(test))
```

```
273

30
```

In [118…

```
# Naive Forecast

naive = test.copy()
naive['naive_forecast'] = train['Sales'][len(train)-1]

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Sales'], label='Train')
plt.plot(test['Sales'], label='Test')
plt.plot(naive['naive_forecast'], label='Naive forecast')
plt.legend(loc='best')
plt.title('Naive Method 30 day forecast')
plt.show()
```



In [104…

```
n_rmse = np.sqrt(mean_squared_error(test['Sales'],
                                    naive['naive_forecast'])).round(2)
n_mape = np.round(np.mean(np.abs(test['Sales']-
                                 naive['naive_forecast'])/
                          test['Sales'])*100,2)

results = pd.DataFrame({'Method':['Naive method 30 day'],
                        'MAPE': [n_mape], 'RMSE': [n_rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[104…

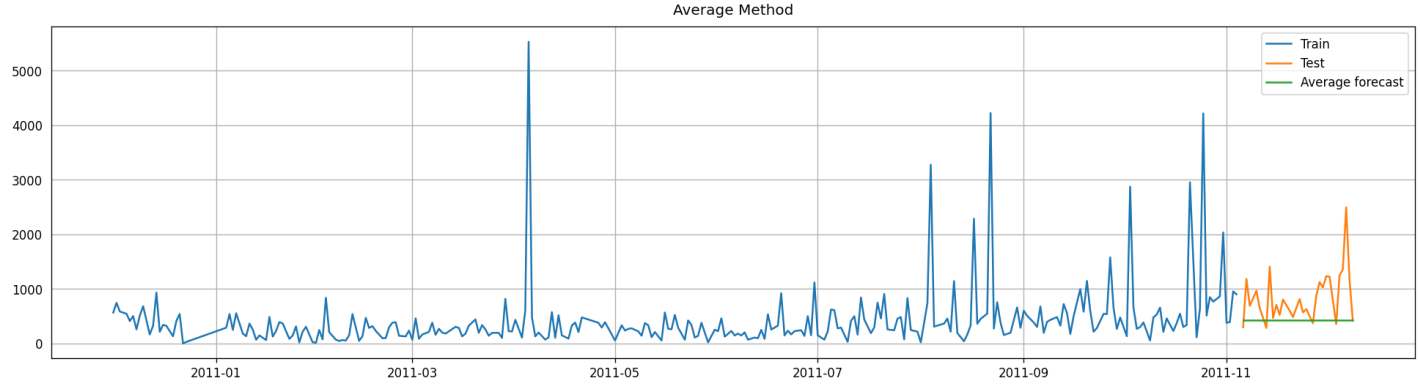
	Method	RMSE	MAPE
0	Naive method	452.17	56.61

Per the graph naive method is not suitable for data with high variability

Simple Average

In [104…

```
simple_average = test.copy()
simple_average['avg_forecast'] = train['Sales'].mean()

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Sales'], label='Train')
plt.plot(test['Sales'], label='Test')
plt.plot(simple_average['avg_forecast'],
         label='Average forecast')
plt.legend(loc='best')
plt.title('Average Method')
plt.show()
```



In [104…

```
sa_rmse = np.sqrt(mean_squared_error(test['Sales'], simple_average['avg_forecast'])).round
sa_mape = np.round(np.mean(np.abs(test['Sales']-
                                  simple_average['avg_forecast'])/
```

```
                                test['Sales'])*100,2)

results = pd.DataFrame({'Method':['Average method'],
                        'MAPE': [sa_mape], 'RMSE': [sa_rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[104…

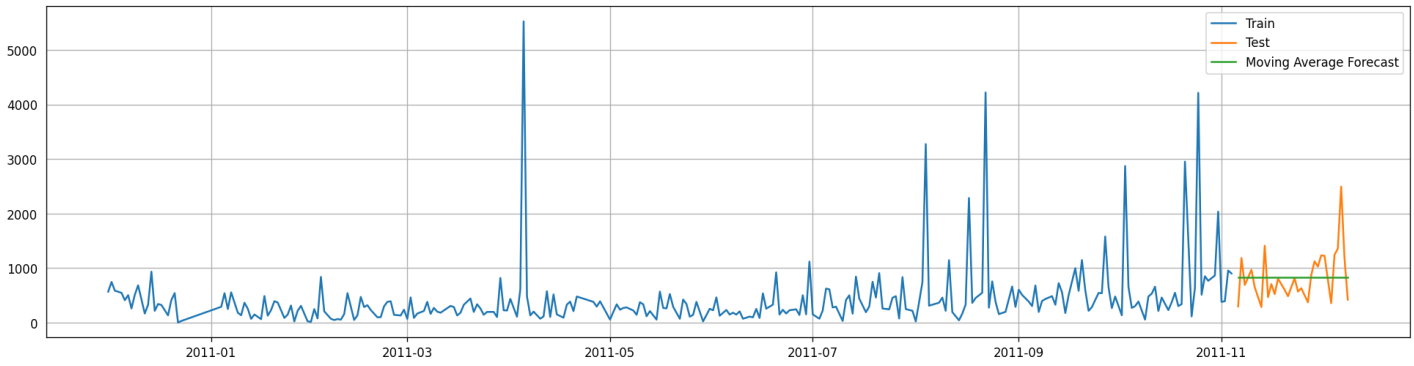
	Method	RMSE	MAPE
0	Average method	613.02	43.52

This model did improve our score, it seems the average of our data is pretty consistent.

Moving Average

In [104…

```
moving_avg = test.copy()
moving_avg['moving_avg_forecast'] =
                        train['Sales'].rolling(29).mean().iloc[-1]

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Sales'], label='Train')
plt.plot(test['Sales'], label='Test')
plt.plot(moving_avg['moving_avg_forecast'],
         label='Moving Average Forecast')
plt.legend(loc='best')
plt.show()
```



In [104…

```
ma_rmse = np.sqrt(mean_squared_error(test['Sales'], moving_avg['moving_avg_forecast'])).ro
ma_mape = np.round(np.mean(np.abs(test['Sales']-
                                  moving_avg['moving_avg_forecast'])/
                           test['Sales'])*100,2)

results = pd.DataFrame({'Method':['Moving Average method'],
                        'MAPE': [ma_mape], 'RMSE': [ma_rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

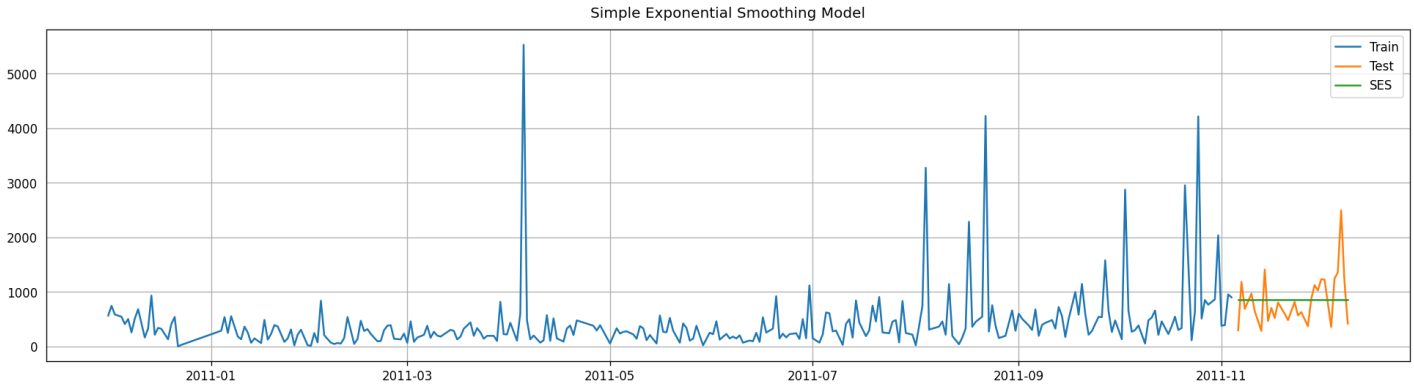
Out[104…

	Method	RMSE	MAPE
0	Moving Average method	449.3	50.82

Interestingly enough this model did not improve our results after choosing the last 60 days. We could adjust the window and see if that improves our results.

Simple Exponential Smoothing

In [129…
```
ses = test.copy()
ses_fit = SimpleExpSmoothing(np.asarray(train['Sales'])).fit(smoothing_level=0.6,
                                                             optimized=False)
ses['SES'] = ses_fit.forecast(len(test))

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Sales'], label='Train')
plt.plot(test['Sales'], label='Test')
plt.plot(ses['SES'], label='SES')
plt.title('Simple Exponential Smoothing Model')
plt.legend(loc='best')
plt.show()
```



In [105…
```
se_rmse = np.sqrt(mean_squared_error(test['Sales'], ses['SES'])).round(2)
se_mape = np.round(np.mean(np.abs(test['Sales']-
                                  ses['SES'])/
                           test['Sales'])*100,2)

results = pd.DataFrame({'Method':['Simple Exponential Smoothing method'],
                        'MAPE': [se_mape],
                        'RMSE': [se_rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

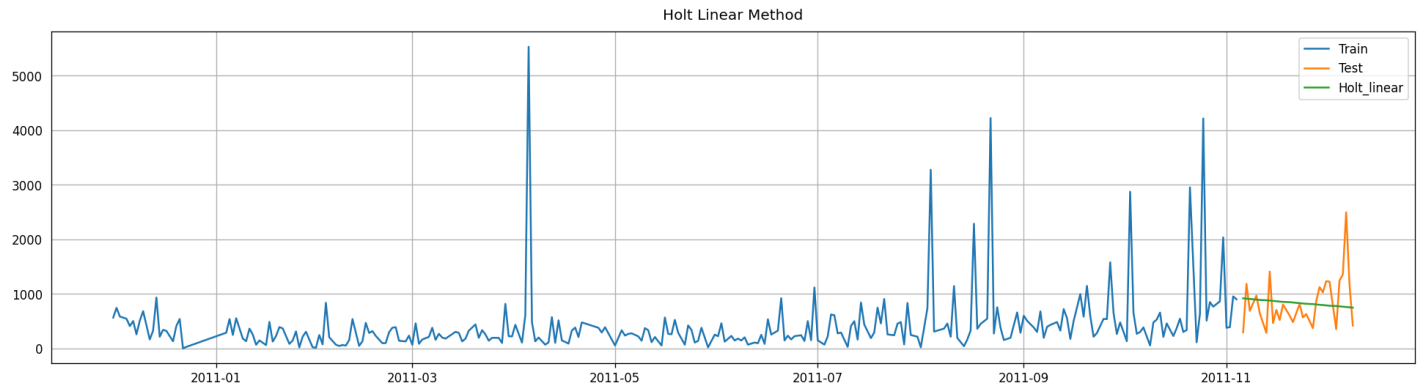
Out[105…

	Method	RMSE	MAPE
0	Simple Exponential Smoothing method	449.3	53.06

So far the second best model after simple average. We can tune to alpha from 0.6 to another number to see if it helps improve the model.

Holt Linear Method (double exponential smoothing)

In [129…
```
holt = test.copy()
holt_fit = Holt(np.asarray(train['Sales'])).fit(smoothing_level = 0.3,
                                                smoothing_slope = 0.1)
holt['Holt_linear'] = holt_fit.forecast(len(test))

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Sales'], label='Train')
plt.plot(test['Sales'], label='Test')
plt.plot(holt['Holt_linear'], label='Holt_linear')
plt.legend(loc='best')
plt.title('Holt Linear Method')
plt.show()
```



In [106…

```
hl_rmse = np.sqrt(mean_squared_error(test['Sales'], holt['Holt_linear'])).round(2)
hl_mape = np.round(np.mean(np.abs(test['Sales']-
                                  holt['Holt_linear'])/
                           test['Sales'])*100,2)

results = pd.DataFrame({'Method':['Holt Linear method'],
                        'MAPE': [hl_mape], 'RMSE': [hl_rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[106…

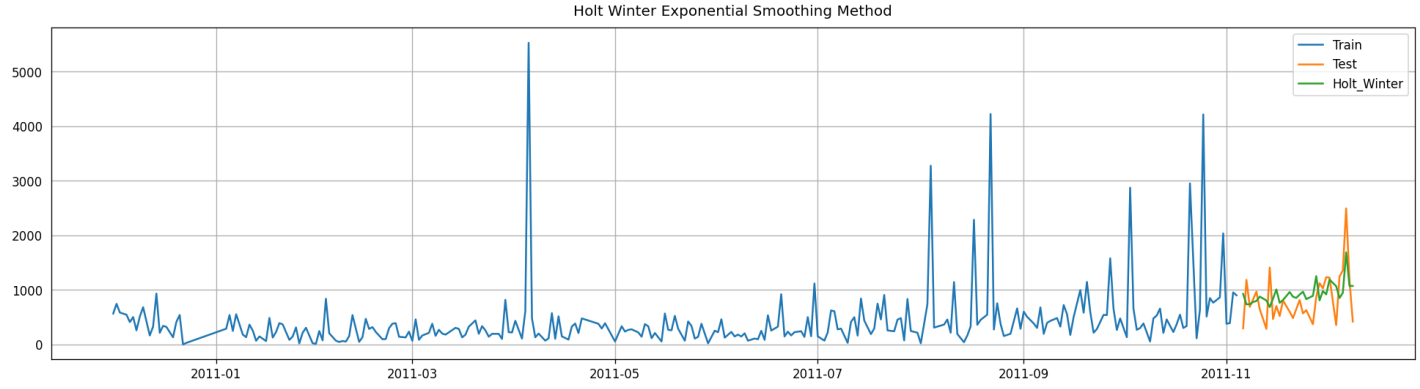
	Method	RMSE	MAPE
0	Holt Linear method	470.65	53.09

Results were not very good on the first run, model can be tuned to see if there's improvement

Holt Winters Method

In [129…

```
hw = test.copy()
hw_fit = ExponentialSmoothing(np.asarray(train['Sales']) ,seasonal_periods=29 ,
                              trend='add',seasonal='add').fit()
hw['Holt_Winter'] = hw_fit.forecast(len(test))

plt.figure(figsize=(20,5))
plt.grid()
plt.plot( train['Sales'], label='Train')
plt.plot(test['Sales'], label='Test')
plt.plot(hw['Holt_Winter'], label='Holt_Winter')
plt.title('Holt Winter Method')
plt.legend(loc='best')
plt.show()
```



In [106…

```
hw_rmse = np.sqrt(mean_squared_error(test['Sales'], hw['Holt_Winter'])).round(2)
hw_mape = np.round(np.mean(np.abs(test['Sales']-
```

```
                                hw['Holt_Winter'])/
                            test['Sales'])*100,2)

results = pd.DataFrame({'Method':['Holt Winters method'],
                        'MAPE': [hw_mape], 'RMSE': [hw_rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

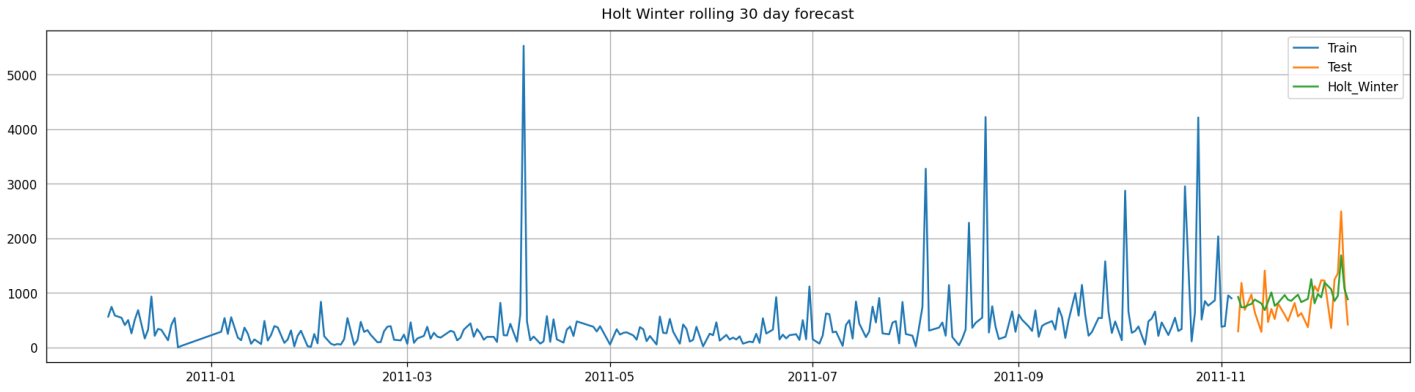
Out[106…

	Method	RMSE	MAPE
0	Holt Winters method	401.7	56.79

Iterative Holt Winters

In [127…

```
# One forecast at a time that is then added to the training
# set and model is recalculated to fit for that new value
# to then repeat a single prediction once again.

hw12 = test.copy()
hw_fit12 = ExponentialSmoothing(np.asarray(train['Sales']) ,seasonal_periods=29 ,
                                trend='add',seasonal='add').fit()
# rolling forecast:
training_set=train.copy()

for ii in range(0,len(test)):
    hw12.loc[hw12.index[ii], 'Holt_Winter'] = hw_fit12.forecast(1)
    training_set.loc[len(training_set.index), 'Sales'] = hw12.loc[hw12.index[ii], 'Holt_Wi
    #training_set_loc[]
    hw_fit12 = ExponentialSmoothing(np.asarray(training_set['Sales']) ,seasonal_periods=29
                                trend='add',seasonal='add').fit()

hw12.loc[hw12.index[ii], 'Holt_Winter'] = hw_fit12.forecast(1)

plt.figure(figsize=(20,5))
plt.grid()
plt.plot( train['Sales'], label='Train')
plt.plot(test['Sales'], label='Test')
plt.plot(hw12['Holt_Winter'], label='Holt_Winter')
plt.legend(loc='best')
plt.title('Holt Winter rolling 30 day forecast')
plt.show()
```



In [127…

```
hw_rmse12 = np.sqrt(mean_squared_error(test['Sales'], hw12['Holt_Winter'])).round(2)
hw_mape12 = np.round(np.mean(np.abs(test['Sales']-hw12['Holt_Winter'])/
                            test['Sales'])*100,2)

results = pd.DataFrame({'Method':['Iterative Holt Winters method 30 day'],
                        'MAPE': [hw_mape12], 'RMSE': [hw_rmse12]})
```

```
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[127…

	Method	RMSE	MAPE
0	Iterative Holt Winters method 30 day	392.78	55.28

Acheived better results with RMSE and the signal forecast seems to follow the general shape of the actual validation data.

ARIMA

First on entire dataset:

In [134…

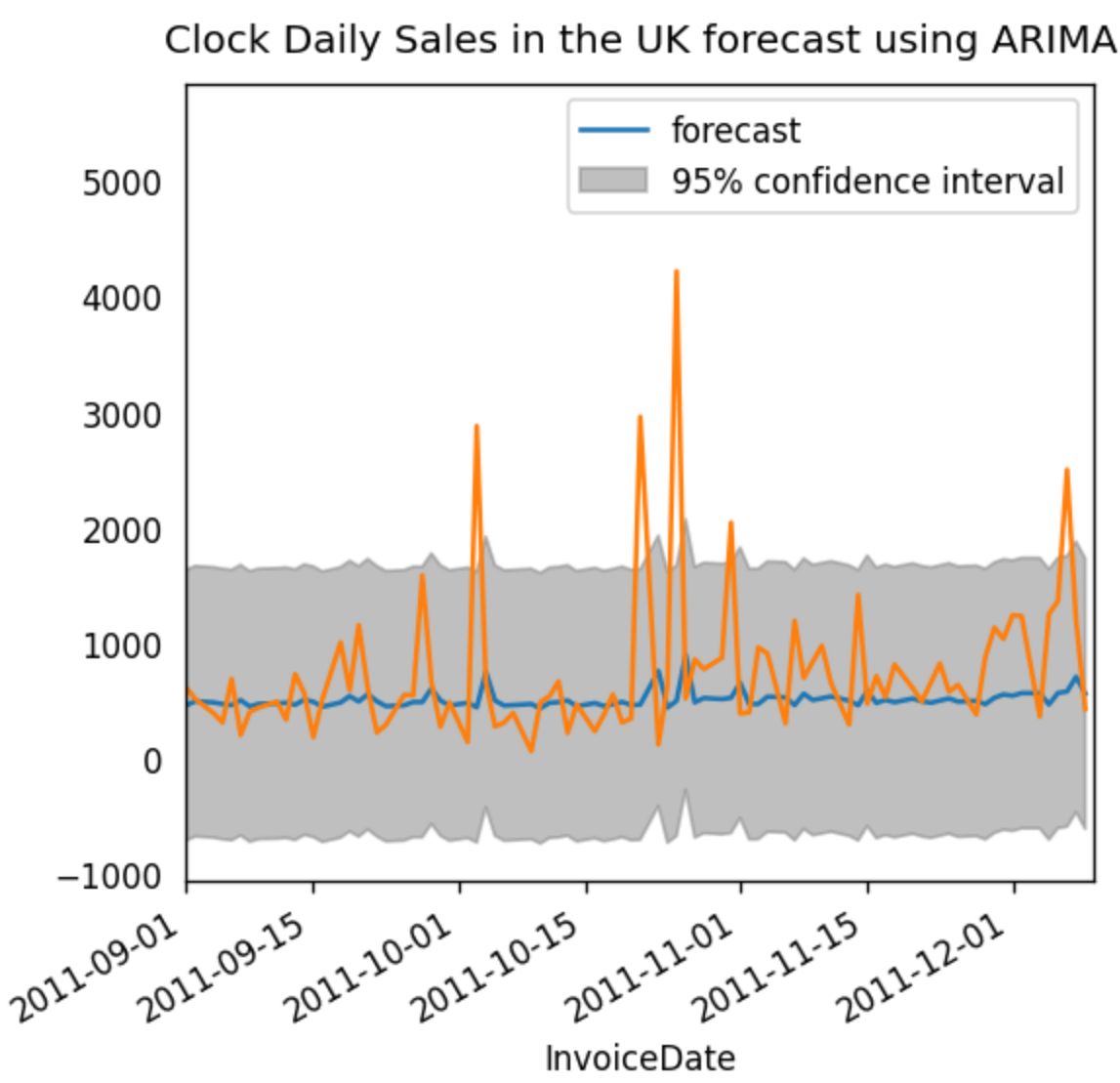
```
# Estimate by trial and error:
# that the best p would be 2 based
# on our autocorrelation plots 1 differencing
# since the signal is stationary, and q = 0
# based on partial autocorrelation plots:

modelclks = ARIMA(UK_DailyClock_df, order=(1,0,0))
model_fitclks = modelclks.fit()
print(model_fitclks.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  Sales   No. Observations:                  303
Model:                 ARIMA(1, 0, 0)   Log Likelihood               -2366.882
Date:                Mon, 05 Dec 2022   AIC                           4739.765
Time:                        11:14:49   BIC                           4750.906
Sample:                             0   HQIC                          4744.222
                                  - 303
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        474.8378     75.449      6.294      0.000     326.961     622.715
ar.L1          0.1118      0.069      1.616      0.106      -0.024       0.247
sigma2       3.58e+05   1.44e+04     24.944      0.000     3.3e+05    3.86e+05
===================================================================================
Ljung-Box (L1) (Q):                   0.01   Jarque-Bera (JB):             11113.26
Prob(Q):                              0.92   Prob(JB):                         0.00
Heteroskedasticity (H):               1.86   Skew:                             4.67
Prob(H) (two-sided):                  0.00   Kurtosis:                        31.16
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

AR(1) on entire dataset gives low AR(1) coefficient, so not a random walk.

In [135…

```
# Actual vs Fitted
plot_predict(model_fitclks)
plt.plot(UK_DailyClock_df)
plt.title('Clock Daily Sales in the UK forecast using ARIMA')
plt.xlim([pd.Timestamp('2011-09-01'), pd.Timestamp('2011-12-10')])
plt.show()
```



Find good ARIMA model:

In [107…
```
UK_DailyClock_df.shape
```

Out[107…
```
(303, 1)
```

In [116…
```
# Create Training and Test
# Forecast last 30 days of series:
train_clk = UK_DailyClock_df.Sales['2010-12-01':'2011-11-04']
test_clk = UK_DailyClock_df.Sales['2011-11-05':]
```

In [116…
```
test_clk.shape
```

Out[116…
```
(30,)
```

Attempt Auto Arima for better parameters:

In [116…
```
model_clk = auto_arima(train, start_p=1, start_q=1,
 test='adf', # use adftest to find optimal 'd'
 max_p=3, max_q=3, # maximum p and q
 m=1, # frequency of series
 d=None, # let model determine 'd'
 seasonal=False, # No Seasonality
 start_P=0,
```

```
 D=0,
 trace=True,
 error_action='ignore',
 suppress_warnings=True,
 stepwise=True)
print(model_clk.summary())
```

```
Performing stepwise search to minimize aic
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=4265.006, Time=0.14 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=4384.993, Time=0.00 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=4344.010, Time=0.03 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=4357.279, Time=0.05 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=4266.882, Time=0.33 sec
 ARIMA(1,0,2)(0,0,0)[0]             : AIC=4266.862, Time=0.33 sec
 ARIMA(0,0,2)(0,0,0)[0]             : AIC=4350.706, Time=0.08 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=4329.391, Time=0.02 sec
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=inf, Time=0.41 sec
 ARIMA(1,0,1)(0,0,0)[0] intercept   : AIC=4275.497, Time=0.06 sec

Best model:  ARIMA(1,0,1)(0,0,0)[0]
Total fit time: 1.471 seconds
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  273
Model:               SARIMAX(1, 0, 1)   Log Likelihood               -2129.503
Date:                Sun, 04 Dec 2022   AIC                           4265.006
Time:                        20:44:36   BIC                           4275.835
Sample:                             0   HQIC                          4269.353
                                  - 273
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.9991      0.004    238.523      0.000       0.991       1.007
ma.L1         -0.9578      0.030    -31.717      0.000      -1.017      -0.899
sigma2      3.444e+05   8906.630     38.664      0.000    3.27e+05    3.62e+05
===================================================================================
Ljung-Box (L1) (Q):                   0.18   Jarque-Bera (JB):             15342.64
Prob(Q):                              0.67   Prob(JB):                         0.00
Heteroskedasticity (H):              18.53   Skew:                             5.35
Prob(H) (two-sided):                  0.00   Kurtosis:                        38.13
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

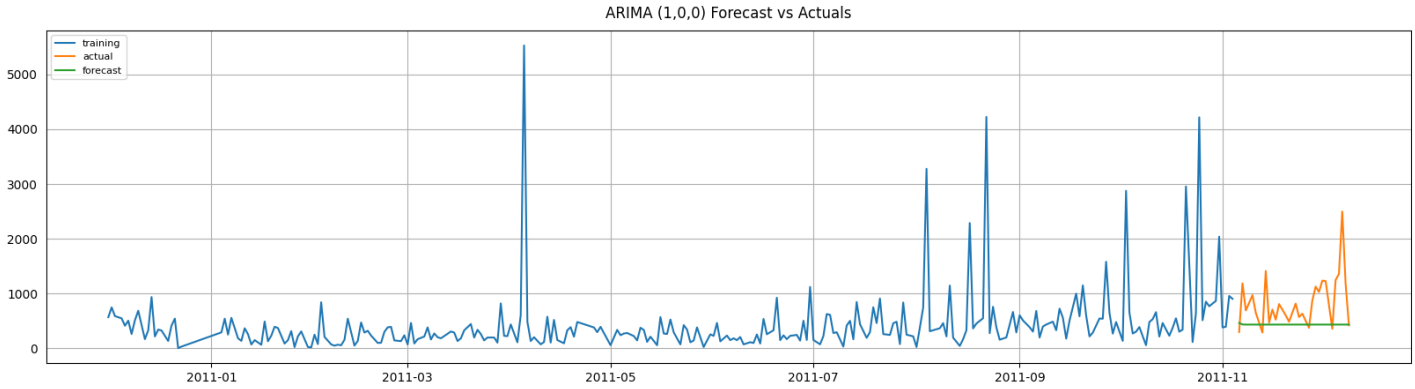
According to auto-arima, the best Arima model using auto-arima is (1,0,1)

ARIMA (1,0,0)

In [126…

```
# Try validating origina (1,0,0) model tried in the
# beginning:

arimaclk_model100 = ARIMA(train, order=(1, 0, 0))
fitted_arimaclk100 = arimaclk_model100.fit()
# Forecast
result_clk100=fitted_arimaclk100.forecast(30, alpha=0.05) # 95% conf

results_indexed100=pd.DataFrame(result_clk100)
results_indexed100['InvoiceDate']=test.index
results_indexed100['InvoiceDate'] = pd.to_datetime(results_indexed100['InvoiceDate'])
results_indexed100=results_indexed100.set_index('InvoiceDate')

## Plot
plt.figure(figsize=(20,5), dpi=100)
```

```
plt.grid()
plt.plot(train, label='training')
plt.plot(test, label='actual')
plt.plot(results_indexed100, label='forecast')
#plt.fill_between(lower_series.index, lower_series, upper_series,
# color='k', alpha=.15)
plt.title('ARIMA (1,0,0) Forecast vs Actuals')
plt.legend(loc='upper left', fontsize=8)
#plt.xlim([pd.Timestamp('2011-10-01'), pd.Timestamp('2011-12-10')])
plt.show()
```



In [117…

```
# Calculating RMSE and MAPE

arima100_rmse = np.sqrt(mean_squared_error(test['Sales'],
                                           results_indexed100['predicted_mean'])).round(2)
arima100_mape = np.round(np.mean(np.abs(test['Sales'] -
                                           results_indexed100['predicted_mean'])/
                                 test['Sales'])*100,2)

results = pd.DataFrame({'Method':['ARIMA method'], 'MAPE': [arima100_mape],
                        'RMSE': [arima100_rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[117…

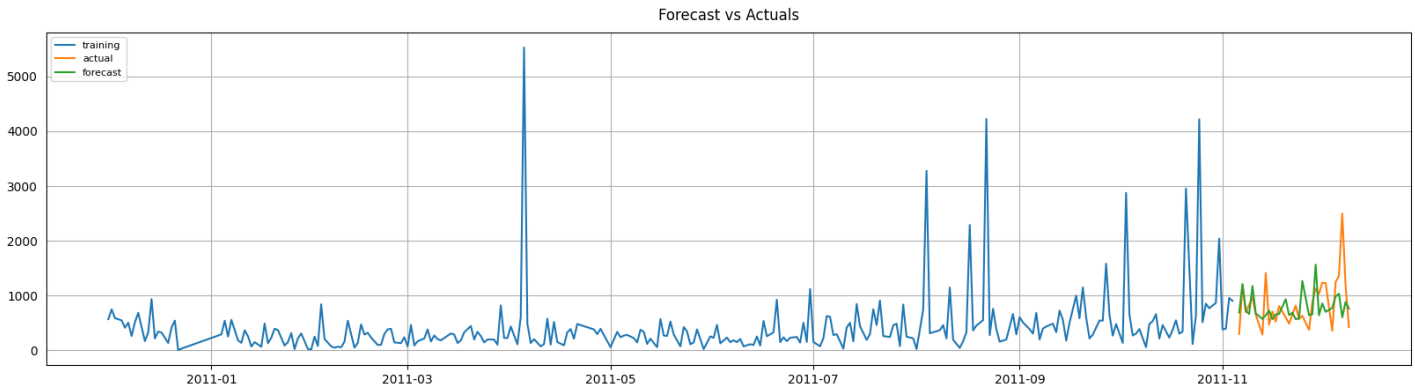
	Method	RMSE	MAPE
0	ARIMA method	613.18	43.85

Using auto arima Recommended model: ARIMA(1,0,1)

In [135…

```
# Since the Holt Winters behaved better with a seasonality
# of about 30, tried this with ARIMA and got better results
# although still not better than Holt Winters method:

arimaclk_model = ARIMA(train, order=(1, 0, 1), seasonal_order=(1,1,1,30))
fitted_arimaclk = arimaclk_model.fit()

# Forecast
result_clk=fitted_arimaclk.forecast(30, alpha=0.05) # 95% conf

results_indexed=pd.DataFrame(result_clk)
results_indexed['InvoiceDate']=test.index
results_indexed['InvoiceDate'] = pd.to_datetime(results_indexed['InvoiceDate'])
results_indexed=results_indexed.set_index('InvoiceDate')

## Plot
plt.figure(figsize=(20,5), dpi=100)
plt.grid()
plt.plot(train, label='training')
```

```
plt.plot(test, label='actual')
plt.plot(results_indexed, label='forecast')
#plt.fill_between(lower_series.index, lower_series, upper_series,
# color='k', alpha=.15)
plt.title('Forecast vs Actuals')
plt.legend(loc='upper left', fontsize=8)
#plt.xlim([pd.Timestamp('2011-10-01'), pd.Timestamp('2011-12-10')])
plt.show()
```



ARIMA (1,0,1)(1,1,1,29) model coefficients

- diplays better AIC than the single (1,0,1) recommendation from the auto_arima function
- all coefficient p-values < 0.05
- std_error is also low

In [135…

```
print(fitted_arimaclk.summary())
```

```
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                               Sales   No. Observations:                  273
Model:             ARIMA(1, 0, 1)x(1, 1, 1, 30)   Log Likelihood               -1922.792
Date:                          Mon, 05 Dec 2022   AIC                           3855.584
Time:                                  11:19:52   BIC                           3873.050
Sample:                                       0   HQIC                          3862.619
                                          - 273
Covariance Type:                            opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.9999      0.024     42.075      0.000       0.953       1.046
ma.L1         -0.9638      0.035    -27.555      0.000      -1.032      -0.895
ar.S.L30       0.1156      0.051      2.256      0.024       0.015       0.216
ma.S.L30      -1.0000      0.031    -32.662      0.000      -1.060      -0.940
sigma2      3.528e+05    8.7e-08   4.06e+12      0.000    3.53e+05    3.53e+05
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):              7289.89
Prob(Q):                              0.98   Prob(JB):                         0.00
Heteroskedasticity (H):               2.19   Skew:                             4.33
Prob(H) (two-sided):                  0.00   Kurtosis:                        28.39
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 9.82e+28. Standa
rd errors may be unstable.
```

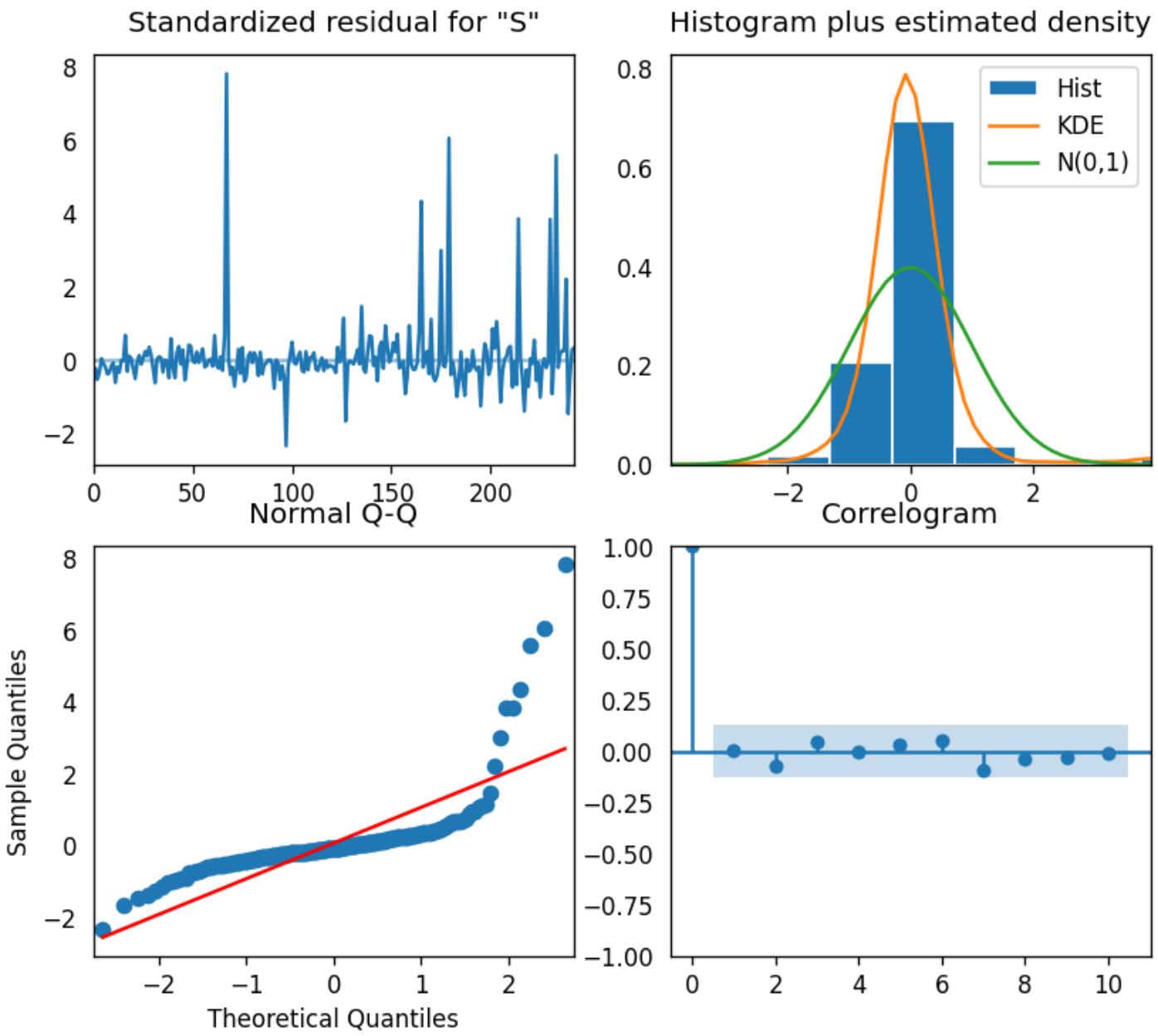
Diagnositcs for ARIMA (1,0,1)(1,1,1,29) model:

- residuals are still with some variability not always centered at 0

- the scatter plot for Sample Quantiles are not as inline with the theoretical as would be desired.

In [135…

```
fitted_arimaclk.plot_diagnostics(figsize=(8,7))
plt.show()
```



In [126…

```
# Calculating RMSE and MAPE

arima_rmse = np.sqrt(mean_squared_error(test['Sales'],
                                        results_indexed['predicted_mean'])).round(2)
arima_mape = np.round(np.mean(np.abs(test['Sales'] -
                                     results_indexed['predicted_mean'])/
                              test['Sales'])*100,2)

results = pd.DataFrame({'Method':['ARIMA method'], 'MAPE': [arima_mape],
                        'RMSE': [arima_rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[126…

	Method	RMSE	MAPE
0	ARIMA method	470.62	42.29

Neural Networks (Long Short-Term Memory Network)

(Brownlee, 2016)

In [763…
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

In [783…
```
UK_DailyClock_df.shape
```

Out[783… (303, 1)

In [784…
```
# Scale data to lie between 0 and 1:

scaler = MinMaxScaler(feature_range=(0, 1))
dataset_neural = scaler.fit_transform(UK_DailyClock_df)
```

To maintain the train/test proportions for other models which have test size of 30, our train/test proportions are 90.2/9.8

In [131…
```
# split into train and test sets
train_size = int(len(dataset_neural) * 0.902)
test_size = len(dataset_neural) - train_size
train_nn, test_nn = dataset_neural[0:train_size,:],
                    dataset_neural[train_size:len(dataset_neural),:]
```

In [131…
```
test_size
```

Out[131… 30

In [131…
```
# convert an array of values and generate
# the X and y for our neural network with
# where X has lagged version of y (the current t)
# and y looks at the future:

def create_dataset(dataset, look_back=1):
 dataX, datay = [], []
 for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        datay.append(dataset[i + look_back, 0])
 return np.array(dataX), np.array(datay)
```

In [132…
```
# Run the above function on the already
# split train and test data so we now have
# X=t and y=t+1

look_back = 1
trainX, trainy = create_dataset(train_nn, look_back)
testX, testy = create_dataset(test_nn, look_back)
```

In [132…
```
# Data has to be reshaped to a format the neural network
```

```
# understands (samples, time steps (1), features)

trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

In [132…

```
# create and fit the LSTM network (original attempt)
n_features_nn=1
n_steps_nn=1
model_nn = Sequential()
model_nn.add(LSTM(4, input_shape=(1, look_back)))
model_nn.add(Dense(1))
model_nn.compile(loss='mean_squared_error', optimizer='adam')
model_nn.fit(trainX, trainy, epochs=100, batch_size=1, verbose=0)
```

Out[132…

```
<keras.callbacks.History at 0x1a2cee151f0>
```

In [ ]:

```
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+
                 look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+
                (look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

In [132…

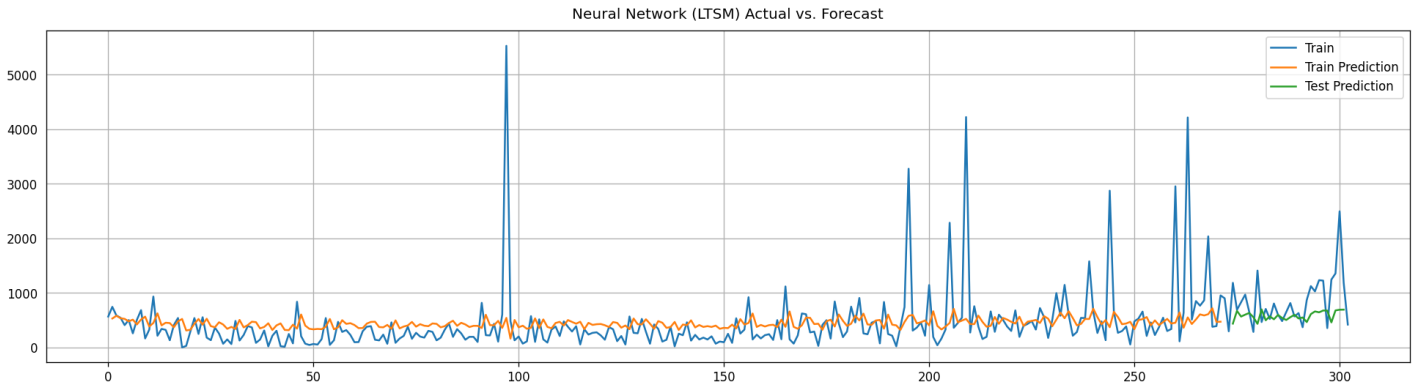
```
# make forecast on both training and test:
trainPredict_nn = model_nn.predict(trainX)
testPredict_nn = model_nn.predict(testX)

# invert scaling done on the data:
trainPredict_nn = scaler.inverse_transform(trainPredict_nn)
trainy = scaler.inverse_transform([trainy])
testPredict_nn = scaler.inverse_transform(testPredict_nn)
testy = scaler.inverse_transform([testy])

# calculate root mean squared error
trainScore_nn = np.sqrt(mean_squared_error(trainy[0],
                                           trainPredict_nn[:,0]))
print('Train Score: %.2f RMSE' % (trainScore_nn))
testScore_nn = np.sqrt(mean_squared_error(testy[0],
                                          testPredict_nn[:,0]))
print('Test Score: %.2f RMSE' % (testScore_nn))
```

```
Train Score: 594.89 RMSE
Test Score: 533.65 RMSE
```

In [132…

```
## make forecast on both training and test:
#trainPredict_nn = model_nn.predict(trainX)
#testPredict_nn = model_nn.predict(testX)

## invert scaling done on the data:
```

```
#trainPredict_nn = scaler.inverse_transform(trainPredict_nn)
#trainy = scaler.inverse_transform([trainy])
#testPredict_nn = scaler.inverse_transform(testPredict_nn)
#testy = scaler.inverse_transform([testy])

# shift train predictions so they can plot
# in line with the original data:
trainPredictPlot_nn = np.empty_like(dataset)
trainPredictPlot_nn[:, :] = np.nan
trainPredictPlot_nn[look_back:len(trainPredict_nn)+
                    look_back, :] = trainPredict_nn

# shift test predictions so they can plot in
# line with the original data:
testPredictPlot_nn = np.empty_like(dataset)
testPredictPlot_nn[:, :] = np.nan
testPredictPlot_nn[len(trainPredict_nn)+(look_back*2)+
                   1:len(dataset)-1, :] = testPredict_nn

# plot baseline and predictions
plt.figure(figsize=(20,5))
plt.grid()
plt.plot(scaler.inverse_transform(dataset_neural),
         label='Train')
plt.plot(trainPredictPlot_nn, label='Train Prediction')
plt.plot(testPredictPlot_nn, label='Test Prediction')
plt.title('Neural Network (LTSM) Actual vs. Forecast')
plt.legend(loc='best')
plt.show()
```



As impressive as Neural Nets can be, and seeing that the forecast does follow any trend however slight in the data, the performance is still lacking when compared to the Holt-Winters model, as was also the case in the Practical Time Series Forecasting example for Chapter 9 (Shmueli & Lichtendahl Jr., 2018)

Linear Regression

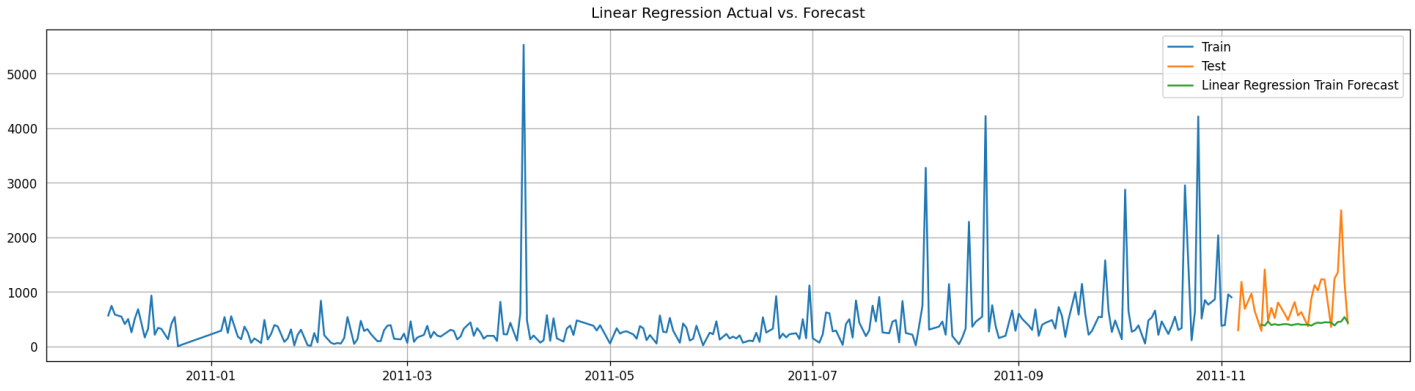
In [133…

```
# This version is not fully comparable to the other methods
# given that the regression is done on a known test data set
# with already known lags, which would normally only be
# available as forecasted lags. However, this is to demonstrate
# even with known data, the linear regression model
# does not seem to outperform many of the other
# data driven models for this series.

train_lr=train.copy()

# Adding lagged versions of data to use
# in liner regression to use as input predictors:

train_lr['Lag_1'] = train_lr['Sales'].shift(1)
```

```
# given their site may have some weak weekly trends
# and they have 6 day weeks:
train_lr['Lag_6'] = train_lr['Sales'].shift(6)


from sklearn.linear_model import LinearRegression

X_lr = train_lr.loc[:, ['Lag_1','Lag_6']]
X_lr.dropna(inplace=True)  # drop missing values in the feature set
y_lr = train_lr.loc[:, 'Sales']  # create the target
y_lr, X_lr = y_lr.align(X, join='inner')  # drop corresponding values in target

model_lr = LinearRegression()
model_lr.fit(X_lr, y_lr)

y_pred_lr_train = pd.Series(model_lr.predict(X_lr),
                            index=X_lr.index)

# Do the same with test data:

test_lr = test.copy()

test_lr['Lag_1'] = test_lr['Sales'].shift(1)
test_lr['Lag_6'] = test_lr['Sales'].shift(6)

X_test_lr = test_lr.loc[:, ['Lag_1','Lag_6']]
X_test_lr.dropna(inplace=True)
y_pred_lr = pd.Series(model_lr.predict(X_test_lr),
                      index=X_test_lr.index)
```

In [133…

```
plt.figure(figsize=(20,5))
plt.grid()
plt.plot( train_lr['Sales'], label='Train')
plt.plot(test_lr['Sales'], label='Test')
plt.plot(y_pred_lr, label='Linear Regression Train Forecast')
plt.title('Linear Regression Actual vs. Forecast')
plt.legend(loc='best')
plt.show()
```



In [133…

```
# Calculating RMSE and MAPE

# because output is for anything after lag of 6,
# need to onlly include data from original dataset
# that represents the last 64 samples instead of 60:

test_lr_error = test_lr.iloc[6:]
lr_rmse = np.sqrt(mean_squared_error(test_lr_error['Sales'],
                                     y_pred_lr)).round(2)
lr_mape = np.round(np.mean(np.abs(test_lr_error['Sales'] -
```

```
                                    y_pred_lr)/
                                test_lr_error['Sales'])*100,2)

results_lr = pd.DataFrame({'Method':['LR method'], 'MAPE': [lr_mape],
                           'RMSE': [lr_rmse]})
results_lr = results_lr[['Method', 'RMSE', 'MAPE']]
results_lr
```

Out[133…

	Method	RMSE	MAPE
0	LR method	651.3	43.61

In [133…

```
# Table Results

Table = PrettyTable(["Model", "RMSE"])
Table.add_row(["Naive", n_rmse])
Table.add_row(["Simple Average", sa_rmse])
Table.add_row(["Moving Average", ma_rmse])
Table.add_row(["Simple Exponential", se_rmse])
Table.add_row(["Holt Linear", hl_rmse])
Table.add_row(["Holt Winter", hw_rmse])
Table.add_row(["Holt Winter iterative", hw_rmse12])
Table.add_row(["ARIMA (1,0,0)", arima100_rmse])
Table.add_row(["ARIMA (1,0,1)(1,1,30)", arima_rmse])
Table.add_row(["Linear Regression", lr_rmse])
Table.add_row(["Neural Network (LSTM)", round(testScore_nn,2)])
print("Time Series Model Performance Sorted by RMSE")
Table.sortby = "RMSE"
print(Table)
```

```
Time Series Model Performance Sorted by RMSE
+-----------------------+--------+
|         Model         |  RMSE  |
+-----------------------+--------+
| Holt Winter iterative | 392.78 |
|      Holt Winter      | 401.7  |
|     Moving Average    | 449.3  |
|   Simple Exponential  | 449.3  |
|         Naive         | 452.17 |
| ARIMA (1,0,1)(1,1,30) | 470.62 |
|      Holt Linear      | 470.65 |
| Neural Network (LSTM) | 533.65 |
|     Simple Average    | 613.02 |
|     ARIMA (1,0,0)     | 613.18 |
|   Linear Regression   | 651.3  |
+-----------------------+--------+
```