**Springboard Time Series - 'Cowboy Cigarettes' Case Study - Tier 1**

**Brief**

You're working in the US federal government as a data scientist in the Health and Environment department. You've been tasked with determining whether sales for the oldest and most powerful producers of cigarettes in the country are increasing or declining.

**Cowboy Cigarettes (TM, *est.* 1890)** is the US's longest-running cigarette manufacturer. Like many cigarette companies, however, they haven't always been that public about their sales and marketing data. The available post-war historical data runs for only 11 years after they resumed production in 1949; stopping in 1960 before resuming again in 1970. Your job is to use the 1949-1960 data to predict whether the manufacturer's cigarette sales actually increased, decreased, or stayed the same. You need to make a probable reconstruction of the sales record of the manufacturer - predicting the future, from the perspective of the past - to contribute to a full report on US public health in relation to major cigarette companies.

The results of your analysis will be used as part of a major report relating public health and local economics, and will be combined with other studies executed by your colleagues to provide important government advice.

As ever, this notebook is **tiered**, meaning you can elect that tier that is right for your confidence and skill level. There are 3 tiers, with tier 1 being the easiest and tier 3 being the hardest.

**1. Sourcing and loading**

* Load relevant libraries
* Load the data
* Explore the data

**2. Cleaning, transforming and visualizing**

* Dropping unwanted columns
* Nomenclature
* Type conversions
* Making a predictor variable y
* Getting summary statistics for y
* Plotting y

**3. Modelling**

* Decomposition
  + Trend
  + Seasonality
  + Noise
* Testing for stationarity with KPSS
* Making the data stationary
* The ARIMA Model
  + Make a function to find the MSE of a single ARIMA model
  + Make a function to evaluate the different ARIMA models with different p, d, and q values
* Visualize the results
* Application: Forecasting

**4. Evaluating and concluding**

* What is our conclusion?
* Next steps

**0. Preliminaries**

Time series data is just any data displaying how a single variable changes over time. It comes as a collection of metrics typically taken at regular intervals. Common examples of time series data include weekly sales data and daily stock prices. You can also easily acquire time series data from [Google Trends](https://trends.google.com/trends/?geo=US), which shows you how popular certain search terms are, measured in number of Google searches.

**1. Sourcing and Loading**

**1a. Load relevant libraries**

import pandas as pd

import numpy as np

import matplotlib.pylab as plt

%matplotlib inline

import warnings

warnings.filterwarnings('ignore')

**1b. Load the data**

Call the variable cigData.

cigData=pd.read\_csv('CowboyCigsData.csv')

**1c. Explore the data**

We now need to check whether the data conduces to a time series style analysis.

cigData.head()

|  | **Unnamed: 0** | **Time** | **#CigSales** |
| --- | --- | --- | --- |
| **0** | 0 | 1949-01 | 1000112 |
| **1** | 1 | 1949-02 | 1000118 |
| **2** | 2 | 1949-03 | 1000132 |
| **3** | 3 | 1949-04 | 1000129 |
| **4** | 4 | 1949-05 | 1000121 |

Over a million cigarettes sold in the month of January 1949. This certainly is a popular cigarette brand.

Check out the columns feature of the data. How many columns are there?

cigData.columns

Index(['Unnamed: 0', 'Time', '#CigSales'], dtype='object')

cigData.shape

(144, 3)

Let's check out the data types of our columns.

cigData.dtypes

Unnamed: 0 int64

Time object

#CigSales int64

dtype: object

Check whether there are any null values.

cigData.isnull().values.any()

False

**2. Cleaning, transforming and visualizing**

**2a. Dropping unwanted columns**

We need to cut that Unnamed: 0 column. Delete it here.

cigData.drop('Unnamed: 0',axis=1)

|  | **Time** | **#CigSales** |
| --- | --- | --- |
| **0** | 1949-01 | 1000112 |
| **1** | 1949-02 | 1000118 |
| **2** | 1949-03 | 1000132 |
| **3** | 1949-04 | 1000129 |
| **4** | 1949-05 | 1000121 |
| **...** | ... | ... |
| **139** | 1960-08 | 1000606 |
| **140** | 1960-09 | 1000508 |
| **141** | 1960-10 | 1000461 |
| **142** | 1960-11 | 1000390 |
| **143** | 1960-12 | 1000432 |

144 rows × 2 columns

**2b. Nomenclature**

We can see that the Time column actually has the granularity of months. Change the name of that column to Month.

cigData.rename(columns = {'Time':'Month'}, inplace = True)

Call a head() to check this has worked.

cigData.head()

|  | **Unnamed: 0** | **Month** | **#CigSales** |
| --- | --- | --- | --- |
| **0** | 0 | 1949-01 | 1000112 |
| **1** | 1 | 1949-02 | 1000118 |
| **2** | 2 | 1949-03 | 1000132 |
| **3** | 3 | 1949-04 | 1000129 |
| **4** | 4 | 1949-05 | 1000121 |

**2c. Type conversions**

Now, do time series analysis on a Pandas dataframe is overkill, and is actually counter-productive. It's much more easy to carry out this type of analysis if we convert our data to a series first.

Notice that the Month field was an object. Let's type convert the Month column to a Python datetime, before making that the index.

cigData.head()

|  | **Unnamed: 0** | **Month** | **#CigSales** |
| --- | --- | --- | --- |
| **0** | 0 | 1949-01 | 1000112 |
| **1** | 1 | 1949-02 | 1000118 |
| **2** | 2 | 1949-03 | 1000132 |
| **3** | 3 | 1949-04 | 1000129 |
| **4** | 4 | 1949-05 | 1000121 |

cigData['Month'] = pd.to\_datetime(cigData['Month'])

cigData.set\_index('Month',inplace=True)

cigData.head()

|  | **Unnamed: 0** | **#CigSales** |
| --- | --- | --- |
| **Month** |  |  |
| **1949-01-01** | 0 | 1000112 |
| **1949-02-01** | 1 | 1000118 |
| **1949-03-01** | 2 | 1000132 |
| **1949-04-01** | 3 | 1000129 |
| **1949-05-01** | 4 | 1000121 |

cigData.drop('Unnamed: 0',axis=1)

|  | **#CigSales** |
| --- | --- |
| **Month** |  |
| **1949-01-01** | 1000112 |
| **1949-02-01** | 1000118 |
| **1949-03-01** | 1000132 |
| **1949-04-01** | 1000129 |
| **1949-05-01** | 1000121 |
| **...** | ... |
| **1960-08-01** | 1000606 |
| **1960-09-01** | 1000508 |
| **1960-10-01** | 1000461 |
| **1960-11-01** | 1000390 |
| **1960-12-01** | 1000432 |

144 rows × 1 columns

Perfect!

**2d. Making a predictor variable y**

The data is now indexed by date, as time series data ought to be.

Since we want to predict the number of cigarette sales at Cowboy cigarettes, and y is typically used to signify a predictor variable, let's create a new variable called y and assign the indexed #Passenger column.

y = cigData['#CigSales']

Check the type of our new variable.

type(y)

pandas.core.series.Series

**2e. Getting summary statistics for y**

Get the summary statistics of our data here.

y.describe()

count 1.440000e+02

mean 1.000280e+06

std 1.199663e+02

min 1.000104e+06

25% 1.000180e+06

50% 1.000266e+06

75% 1.000360e+06

max 1.000622e+06

Name: #CigSales, dtype: float64

Try visualizing the data. A simple matplotlib plot should do the trick.

**2f. Plotting y**

y.plot()

<matplotlib.axes.\_subplots.AxesSubplot at 0x211cf335b50>

A blue line graph on a white background

Description automatically generated

**3. Modelling**

**3a. Decomposition**

What do you notice from the plot? Take at least 2 minutes to examine the plot, and write down everything you observe.

All done?

We can see that, generally, there is a trend upwards in cigarette sales from at Cowboy Cigarettes. But there are also some striking - and perhaps unexpected - seasonal fluctuations. These seasonal fluctations come in a repeated pattern. Work out when these seasonal fluctuations are happening, and take 2 minutes to hypothesize on their cause here.

What does it mean to *decompose* time series data? It means breaking that data into 3 components:

1. **Trend**: The overall direction that the data is travelling in (like upwards or downwards)
2. **Seasonality**: Cyclical patterns in the data
3. **Noise**: The random variation in the data

We can treat these components differently, depending on the question and what's appropriate in the context. They can either be added together in an *additive* model, or multiplied together in a *multiplicative* model.

Make a coffee, take 5 minutes and read [this article](https://medium.com/@sigmundojr/seasonality-in-python-additive-or-multiplicative-model-d4b9cf1f48a7) and think about whether our data would conduce to an additive or multiplicative model here. Write your conclusion down just here:

All done? Well, just on the basis of the plot above, it seems our Cowboy Cigarettes data is actually multiplicative.

That's because, as time progresses, the general trend seems to be increasing *at a rate that's also increasing*. We also see that the seasonal fluctuations (the peaks and troughs) get bigger and bigger as time progresses.

Now on the other hand, if the data were simply additive, we could expect the general trend to increase at a *steadily*, and a constant speed; and also for seasonal ups and downs not to increase or decrease in extent over time.

Happily, we can use the decompose() function to quantify the component parts described above in our data.

# Import seasonal\_decompose

from statsmodels.tsa.seasonal import seasonal\_decompose

# Make a variable called decomposition, and assign it y passed to seasonal\_decompose()

decomposition= seasonal\_decompose(y)

# Make three variables for trend, seasonal and residual components respectively.

# Assign them the relevant features of decomposition

trend = decomposition.trend

seasonal = decomposition.seasonal

residual = decomposition.resid

# Plot the original data, the trend, the seasonality, and the residuals

plt.subplot(411)

plt.plot(y, label = 'original data')

plt.legend(loc = 'best')

plt.subplot(412)

plt.plot(trend, label = 'Trend')

plt.legend(loc = 'best')

plt.subplot(413)

plt.plot(seasonal, label = 'seasonality')

plt.legend(loc = 'best')

plt.subplot(414)

plt.plot(residual, label = 'Residuals')

plt.legend(loc = 'best')

plt.tight\_layout()

A screenshot of a graph

Description automatically generated

**3b. Testing for stationarity with KPSS**

As you know, when doing time series analysis we always have to check for stationarity. Imprecisely, a time series dataset is stationary just if its statistical features don't change over time. A little more precisely, a stationary time series dataset will have constant mean, variance, and covariance.

There are many ways to test for stationarity, but one of the most common is the KPSS test. The Null hypothesis of this test is that the time series data in question is stationary; hence, if the *p*-value is less than the significance level (typically 0.05, but we decide) then we reject the Null and infer that the data is not stationary.

from statsmodels.tsa.stattools import kpss

# Use kpss()

kpss(y)

(1.0521750110138661,

0.01,

14,

{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739})

Since our p-value is less than 0.05, we should reject the Null hypothesis and deduce the non-stationarity of our data.

But our data need to be stationary! So we need to do some transforming.

**3c. Making the data stationary**

Let's recall what it looks like.

y.plot()

<matplotlib.axes.\_subplots.AxesSubplot at 0x211d1cc4520>

A blue line graph on a white background

Description automatically generated

In our plot, we can see that both the mean and the variance *increase as time progresses*. At the moment, our data has neither a constant mean, nor a constant variance (the covariance, however, seems constant).

One ofte used way of getting rid of changing variance is to take the natural log of all the values in our dataset. Let's do this now.

# Declare a variable called y\_log

y\_log = np.log(y)

When you plot this, you can see how the variance in our data now remains contant over time.

y\_log.plot()

<matplotlib.axes.\_subplots.AxesSubplot at 0x211d1d20a90>

A blue line graph on a white background

Description automatically generated

We now have a constant variance, but we also need a constant mean.

We can do this by *differencing* our data. We difference a time series dataset when we create a new time series comprising the difference between the values of our existing dataset.

Python is powerful, and we can use the diff() function to do this. You'll notice there's one less value than our existing dataset (since we're taking the difference between the existing values).

y\_diff=y\_log.diff().dropna()

y\_diff.plot()

<matplotlib.axes.\_subplots.AxesSubplot at 0x211d1e3c820>

A blue line on a white background

Description automatically generated

kpss(y\_diff)

C:\Users\Admin\anaconda3\lib\site-packages\statsmodels\tsa\stattools.py:1687: InterpolationWarning: p-value is greater than the indicated p-value

warn("p-value is greater than the indicated p-value", InterpolationWarning)

(0.0530107985985768,

0.1,

14,

{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739})

Our p-value is now greater than 0.05, so we can accept the null hypothesis that our data is stationary.

**3d. The ARIMA model**

Recall that ARIMA models are based around the idea that it's possible to predict the next value in a time series by using information about the most recent data points. It also assumes there will be some randomness in our data that can't ever be predicted.

We can find some good parameters for our model using the sklearn and statsmodels libraries, and in particular mean\_squared\_error and ARIMA.

# Import mean\_squared\_error and ARIMA

from sklearn.metrics import mean\_squared\_error

from statsmodels.tsa.arima\_model import ARIMA

**3di. Make a function to find the MSE of a single ARIMA model**

# Make a function called evaluate\_arima\_model to find the MSE of a single ARIMA model

def evaluate\_arima\_model(data, arima\_order):

# Needs to be an integer because it is later used as an index.

# Use int()

split=int(len(data) \* 0.8)

# Make train and test variables, with 'train, test'

train, test = data[0:split], data[split:len(data)]

past=[x for x in train]

# make predictions

predictions = list()

for i in range(len(test)):#timestep-wise comparison between test data and one-step prediction ARIMA model.

model = ARIMA(past, order=arima\_order)

model\_fit = model.fit(disp=0)

future = model\_fit.forecast()[0]

predictions.append(future)

past.append(test[i])

# calculate out of sample error

error = mean\_squared\_error(test, predictions)

# Return the error

return error

**3dii. Make a function to evaluate the different ARIMA models with different p, d, and q values**

# Make a function called evaluate\_models to evaluate different ARIMA models with several different p, d, and q values.

def evaluate\_models(dataset, p\_values, d\_values, q\_values):

best\_score, best\_cfg = float("inf"), None

# Iterate through p\_values

for p in p\_values:

# Iterate through d\_values

for d in d\_values:

# Iterate through q\_values

for q in q\_values:

# p, d, q iterator variables in that order

order = (p,d,q)

try:

# Make a variable called mse for the Mean squared error

mse = evaluate\_arima\_model(dataset, order)

if mse < best\_score:

best\_score, best\_cfg = mse, order

print('ARIMA%s MSE=%.3f' % (order,mse))

except:

continue

return print('Best ARIMA%s MSE=%.3f' % (best\_cfg, best\_score))

# Now, we choose a couple of values to try for each parameter: p\_values, d\_values and q\_values

# Fill in the blanks as appropriate

p\_values= [x for x in range(0, 3)]

d\_values = [x for x in range(0, 3)]

q\_values = [x for x in range(0, 3)]

# Finally, we can find the optimum ARIMA model for our data.

# Nb. this can take a while...!

import warnings

warnings.filterwarnings("ignore")

evaluate\_models(y\_log, p\_values, d\_values, q\_values)

ARIMA(0, 0, 0) MSE=0.000

ARIMA(0, 0, 1) MSE=0.000

ARIMA(0, 1, 0) MSE=0.000

ARIMA(0, 1, 1) MSE=0.000

ARIMA(0, 1, 2) MSE=0.000

ARIMA(0, 2, 0) MSE=0.000

ARIMA(0, 2, 1) MSE=0.000

ARIMA(0, 2, 2) MSE=0.000

ARIMA(1, 0, 0) MSE=0.000

ARIMA(1, 0, 1) MSE=0.000

ARIMA(1, 0, 2) MSE=0.000

ARIMA(1, 1, 0) MSE=0.000

ARIMA(1, 1, 1) MSE=0.000

ARIMA(1, 2, 0) MSE=0.000

ARIMA(2, 0, 0) MSE=0.000

ARIMA(2, 0, 1) MSE=0.000

ARIMA(2, 1, 0) MSE=0.000

ARIMA(2, 1, 1) MSE=0.000

ARIMA(2, 1, 2) MSE=0.000

ARIMA(2, 2, 0) MSE=0.000

Best ARIMA(2, 1, 1) MSE=0.000

So the best p,d, q, parameters for our ARIMA model are 2, 1, 1 respectively. Now we know this, we can build the model.

p=2

d=1

q=1

model = ARIMA(y\_log, order=(p,d,q))

model\_fit = model.fit()

forecast = model\_fit.forecast(24)

We can take a look at a summary of the model this library has built around our data.

# Call summary() on model\_fit

model\_fit.summary()

|  |  |  |  |
| --- | --- | --- | --- |
| ARIMA Model Results  **Dep. Variable:** | D.#CigSales | **No. Observations:** | 143 |
| **Model:** | ARIMA(2, 1, 1) | **Log Likelihood** | 1263.116 |
| **Method:** | css-mle | **S.D. of innovations** | 0.000 |
| **Date:** | Tue, 13 Oct 2020 | **AIC** | -2516.233 |
| **Time:** | 15:58:28 | **BIC** | -2501.418 |
| **Sample:** | 02-01-1949 | **HQIC** | -2510.213 |
|  | - 12-01-1960 |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **coef** | **std err** | **z** | **P>|z|** | **[0.025** | **0.975]** |
| **const** | 2.624e-06 | 5.06e-07 | 5.184 | 0.000 | 1.63e-06 | 3.62e-06 |
| **ar.L1.D.#CigSales** | 0.4681 | 0.156 | 3.003 | 0.003 | 0.163 | 0.774 |
| **ar.L2.D.#CigSales** | -0.2640 | 0.109 | -2.413 | 0.016 | -0.478 | -0.050 |
| **ma.L1.D.#CigSales** | -0.8693 | nan | nan | nan | nan | nan |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Roots | **Real** | **Imaginary** | **Modulus** | **Frequency** |
| **AR.1** | 0.8866 | -1.7326j | 1.9463 | -0.1747 |
| **AR.2** | 0.8866 | +1.7326j | 1.9463 | 0.1747 |
| **MA.1** | 1.1504 | +0.0000j | 1.1504 | 0.0000 |

**3e. Visualize the results**

Visualize the original dataset plotted against our model.

# Call figure() and plot() on the plt

plt.figure(figsize=(15,10))

plt.plot(y\_log.diff())

plt.plot(model\_fit.predict(), color = 'red')

[<matplotlib.lines.Line2D at 0x1cec50f8eb0>]

A blue and red line

Description automatically generated

**3f. Application: Forecasting**

We've done well: our model fits pretty closely to our existing data. Let's now use it to forecast what's likely to occur in future.

# Declare a variable called forecast\_period with the amount of months to forecast, and

# create a range of future dates that is the length of the periods you've chosen to forecast

forecast\_period= 12

date\_range = pd.date\_range(y\_log.index[-1], periods = forecast\_period,

freq='MS').strftime("%Y-%m-%d").tolist()

# Convert that range into a dataframe that includes your predictions

# First, call DataFrame on pd

future\_months = pd.DataFrame(date\_range, columns = ['Month'])

# Let's now convert the 'Month' column to a datetime object with to\_datetime

future\_months['Month'] = pd.to\_datetime(future\_months['Month'])

future\_months.set\_index('Month', inplace = True)

future\_months['Prediction'] = pd.Series(forecast[0])

# Plot your future predictions

# Call figure() on plt

plt.figure(figsize=(15,10))

plt.plot(y\_log)

plt.plot(y\_log['Nov 1960'].append(future\_months['Prediction']))

plt.show()

A graph of a graph

Description automatically generated

# Now plot the original variable y

# Use the same functions as before

plt.figure(figsize=(15,10))

plt.plot(y)

plt.plot(np.exp(y\_log['Nov 1960'].append(future\_months['Prediction'])))

plt.show()

A graph of a graph

Description automatically generated

**4. Evaluating and Concluding**

Our model captures the centre of a line that's increasing at a remarkable rate. Cowboy Cigarettes sell more cigarettes in the summer, perhaps due to the good weather, disposable income and time off that people enjoy, and the least in the winter, when people might be spending less and enjoying less free time outdoors.

Remarkably, our ARIMA model made predictions using just one variable. We can only speculate, however, on the causes of the behaviour predicted by our model. We should also take heed that spikes in data, due to sudden unusual circumstances like wars, are not handled well by ARIMA; and the outbreak of the Vietnam War in the 1960s would likely cause our model some distress.

We could suggest to our employers that, if they are interested in discovering the causes of the cigarette sales trajectory, they execute a regression analysis in addition to the time series one.