Contents

1. Process Outline

1. **Load the data.**

Part 1

2. **Pre-processing:** Creating timestamps, converting the datatype of date/time column, making the series univariate, etc.
3. **Check and make the series stationary:** Most time series methods assume the data is stationary. Therefore, we need to check that using tests like ADF and perform transformations like differencing to make the series stationary.
4. **Differencing (If needed):** The number of times a difference operation needs to be performed to make the series stationary is the **d** value.

   Note: Steps 3 and 4 may not always be necessary. FB Prophet and Auto ARIMA can handle this by themselves.

Part 2

5. Univariate or Multivariate?
6. **Create ACF and PACF plots:** ACF and PACF plots are used to determine the input parameters **p and q** for the ARIMA model.
7. **Determine the p and q values:** Read the values of p and q from the plots in the previous step.

   Note: Steps 6 and 7, like 3 and 4 are also not required in certain cases. Auto ARIMA can find the values by itself while methods like VAR and LSTM have different ways of operating.
8. **Fit the model of choice:** Using the processed data and parameter values we calculated from the previous steps to fit the ARIMA model. Other methods like Prophet, LSTM and VAR have different procedures which will be covered in the next sections.

Part 3

9. **Predict values on validation set:** Predict the future values and compare with the test data.
10. **Evaluate the forecasts:** To check the performance of the model, check the RMSE value using the predictions and actual values on the validation set.

2. Preprocessing (Part 1)

Sections 1 to 4 in the accompanying notebooks contain the code for the processes explained in this section.

Section 1: Importing packages.
Section 2: Loading data, handling NULLs and parsing dates.
Section 3: Setting dates as index, train/test split, resampling.
Section 4: Visualizing and stationary tests.

Check Nulls

Check for null values in the dataset. The below command will show us how many null values are in each column of the dataset.

```
perthTemp.isnull().sum()
```

```
Year                               1
Month                              1
Day                                1
Minimum temperature (Degree C)    76
Maximum temperature (Degree C)    31
Rainfall amount (millimetres)      0
dtype: int64
```

Missing values

One of the simplest ways to fill missing values is to use the previous or next value. This method works well enough in most of the cases as time series data usually changes gradually. As long as there aren't multiple NULLs in a row, we can use backward or forward fills. The code below is a method in pandas that does this.

```
perthTemp.fillna(method='ffill', inplace = True)
```

method is an argument that specifies how the null position is filled. It takes the following values –

- pad / ffill: propagate the last valid observation forward to the next field.
- backfill / bfill: use the next valid observation to fill the gap.

Click reference [1] below for more info.

Timestamps

Parsing Dates

Datasets come with dates in many different formats. It could be in a single column like in the first image below or it could be split into multiple columns like in the second image.

date
1944-06-03
1944-06-04
1944-06-05
1944-06-06

Year	Month	Day
1944.0	6.0	3.0
1944.0	6.0	4.0
1944.0	6.0	5.0
1944.0	6.0	6.0

We can combine the 3 columns into a single column while reading the CSV using pandas.

```
test = pd.read_csv("data/PerthTemperatures.csv",
                   parse_dates=[['Year', 'Month', 'Day']])
```

This will combine the output as shown below.

Year_Month_Day
1944 6 3
1944 6 4
1944 6 5
1944 6 6

Pandas usually reads date column as strings when reading the CSV file. That needs to be converted to datetime datatype. This can be done using the following code.

```
dataframe['Date'] = pd.to_datetime(dataframe[['Year', 'Month',
                                              'Day']])
```

We can check if the date column is of the right data type using the code below.

```
dataframe.dtypes
```

Output:

```
Minimum temperature (Degree C)          float64
Maximum temperature (Degree C)          float64
Rainfall amount (millimetres)           float64
Date                             datetime64[ns]
dtype: object
```

Date index

Setting the date time column as the index for the data frame makes it very convenient to handle.

We can split the dataset into train test sets based on the date and performing any transformations or operations on the entire dataframe becomes easy when the date column in out of the way. We can do that using the code below.

```
dataframe = dataframe.set_index('Date')
```

Output:

	Minimum temperature (Degree C)
Date	
1944-06-03	11.0
1944-06-04	12.2
1944-06-05	12.0
1944-06-06	7.4

Resample

Time series data can be hourly, daily, monthly, or even once every second or any frequency. The code below can be used to resample the data into a different frequency based on the requirements.

```
Dataframe = dataframe.resample('MS').mean()
```

Train/Test Split

We can use the time index to easily split the dataset.

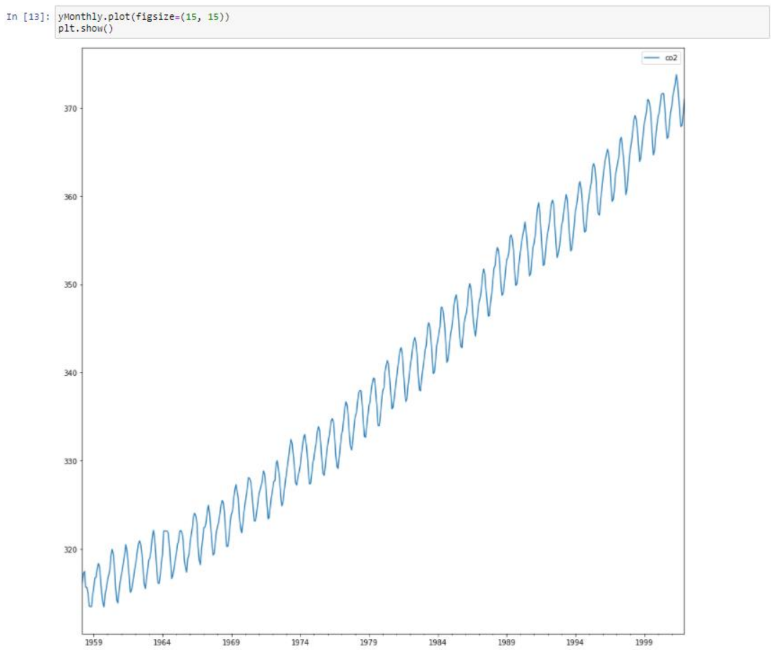
```
dataframe_Train = dataframe[:'2010-12-31']

dataframe_Test = dataframe['2011-01-01':]
```

3. Visualizing Data

The first step in Time Series Analysis is visualizing the data. A simple plot can give us the basic information. We can use matplotlib for basic plotting.

```
dataframe.plot(figsize = (15, 15))
plt.show()
```

Output:



Stationarity Tests

A stationary time series is one whose statistical properties like mean and variance are constant over time. Most statistical forecasting methods assume that the time series data is stationary. Therefore, it is important to check if the data is non-stationary.

One of the most popular methods for checking stationarity is **Augmented Dickey-Fuller test.**

Null Hypothesis (H0): . It has some time dependent structure. If failed to be rejected, it suggests the time series is non-stationary.

Alternate Hypothesis (H1): The null hypothesis is rejected; it suggests the time series is stationary. It does not have time-dependent structure.

We interpret this result using the **p-value** from the test. A p-value below a threshold (such as 5% or 1%) suggests we reject the null hypothesis (therefore the data is stationary), otherwise a p-value above the threshold suggests we fail to reject the null hypothesis (thus the data is non-stationary).

Below code shows a function that accepts a time series and does a stationarity test

```python
def adf_test(ts, signif=0.05):
    dftest = adfuller(ts, autolag='AIC')
    adf = pd.Series(dftest[0:4], index=['Test Statistic','p-value','# Lags','# Observations'])

    for key,value in dftest[4].items():
       adf['Critical Value (%s)'%key] = value
    print (adf)

    p = adf['p-value']
    if p <= signif:
        print(f" Series is Stationary")
    else:
        print(f" Series is Non-Stationary")
```

Output of the test

```
Test Statistic            2.359810
p-value                   0.998990
# Lags                   14.000000
# Observations          511.000000
Critical Value (1%)      -3.443212
Critical Value (5%)      -2.867213
Critical Value (10%)     -2.569791
dtype: float64
 Series is Non-Stationary
```

How to make a time series stationary?

A time series can be thought of as a combination of **Level, Trend, Seasonality and Noise**

Level – The average value of the series

Trend - The increasing or decreasing value in the series

Seasonality – Repeating cycles in the series

Noise – Random variation in the series

Trend and Seasonality are what make a time series non stationary. There are two common ways to deal with trend and stationarity and make the time series stationary. **Differencing and Decomposition**

Decomposing a Timeseries

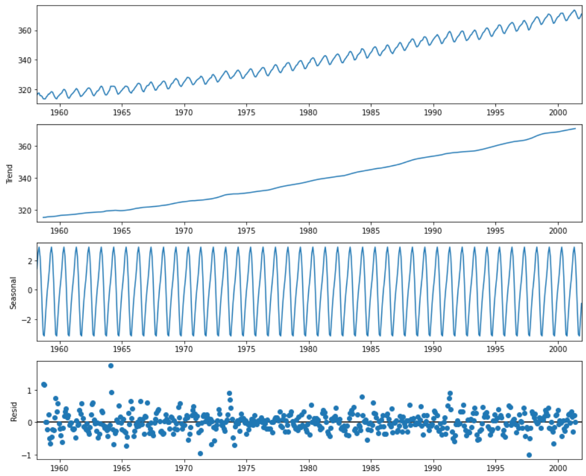
We can decompose the given time series as shown below.

```python
decomposition = sm.tsa.seasonal_decompose(yMonthly,
model='additive')
```

```
fig = decomposition.plot()
plt.show()
```

The individual components can be accessed from the returned object. They can simply be removed from the original time series to make it stationary.

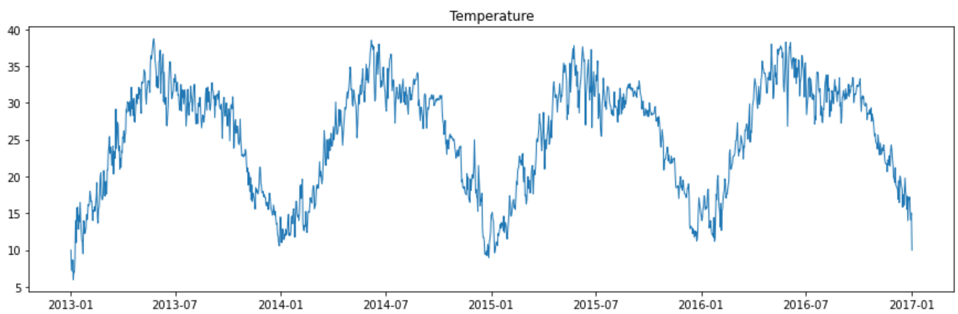
```
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid
```



Differencing

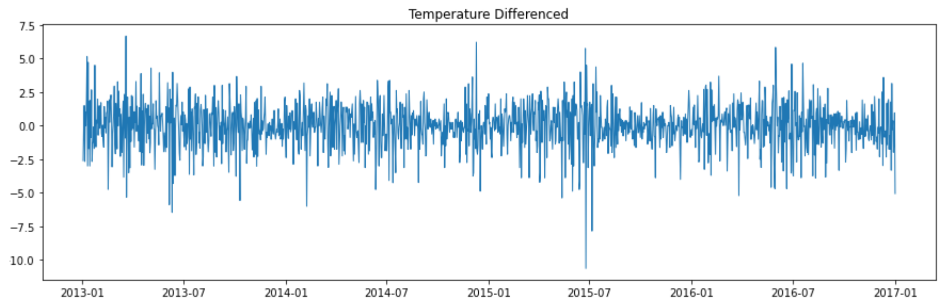
The first difference of a time series is the series of changes from one period to the next.

The image below shows the original values of a time series.



```
Differenced_dataframe = dataframe.diff()
```

The image below shows the first difference of the above time series.



However, if we are only dealing with **univariate** time series analysis, we **need not do the differencing manually**. That will be handled by the forecasting models discussed in the next section.

For **Multivariate** analysis we will have to **do the differencing manually** and invert the process after the forecasting to get the correct values. **Manual differencing and inverting** will be explained in section 5 of this report.

For more information check the reference number [2] below.

Part 2 – Picking the model

If there is only one column of data for which we are trying to forecast the future values, it is called univariate time series analysis. One of the most popular models for univariate analysis is **ARIMA.**

If there are multiple columns of data and we are trying to forecast all of them and they all influence each other , we call it multivariate analysis. **Vector Auto Regression** is one of the popular multivariate analysis models.

If we are only interested in forecasting one target variable but we want to use multiple input or predictor variables we can use **Facebook's Prophet.**

LSTMs are a variant of Recurring Neural Networks that can be used to model univariate or multivariate forecasting problems. They require slightly different pre-processing of data to be trained.

4. Univariate Analysis

ARIMA Model

The notebooks **ARIMA_Univariate** and **ARIMA_NonStationary** contain the full code that is explained below. The notebook named Non Stationary contains an example with a non-stationary timeseries

ARIMA stands for **Auto Regressive Integrated Moving Average**. An ARIMA model has 3 components.

AR – Autoregression refers to the part of the model that regresses on its own lags (previous values). To put it simply the forecast value is a weighted sum of one or more previous values of the variable being forecasted.

MA – Moving Average model fits on the errors from lagged observations. The forecast values are a weighted sum of one or more previous values of errors. The error here is the difference between the actual observation and the value predicted by the moving average model.

I – If the time series is non-stationary it needs to be differenced and it is said to be an integrated version of the series.

The basic code to fit a model and generate a forecast is given below.

```
model = ARIMA(data_frame[['column_name']], order=(p,d,q))

model_fit = model.fit()

fc = model_fit.forecast(len(test_data), alpha=0.05)

fc_series = pd.Series(fc, index =
                              test_data[['column']].index)
```

The ARIMA model has 3 main parameters **p, q, d,** which are passed to the model through the **order** argument seen above.

p = Periods to lag (AR Terms). If P = 3 then we will use the three previous periods of our time series in the autoregressive portion of the calculation

q = This is the number of MA terms. These are lagged forecast errors. If q is 5, the predictors for x(t) will be e(t-1)….e(t-5) where e(i) is the difference between the moving average at ith instant and actual value. This variable denotes the lag of the error component, where error component is a part of the time series not explained by trend or seasonality

d = d refers to the number of differencing transformations required by the time series to get stationarity.

Determining the Parameters for ARIMA

To use ARIMA for forecasting we need to determine the **p, q, d parameters.** Refer section 5 in the ARIMA notebooks.

Differencing parameter (d)

The differencing parameters is the easiest to find. We just use the **Augmented Dickey Fuller** test explained above to check the stationarity of each series and difference the values using the code shown in the previous sections. After the differencing is done, we perform the test again and keep differencing the time series until it is stationary.

Autocorrelation and Partial Autocorrelation Plots (Determining p and q)

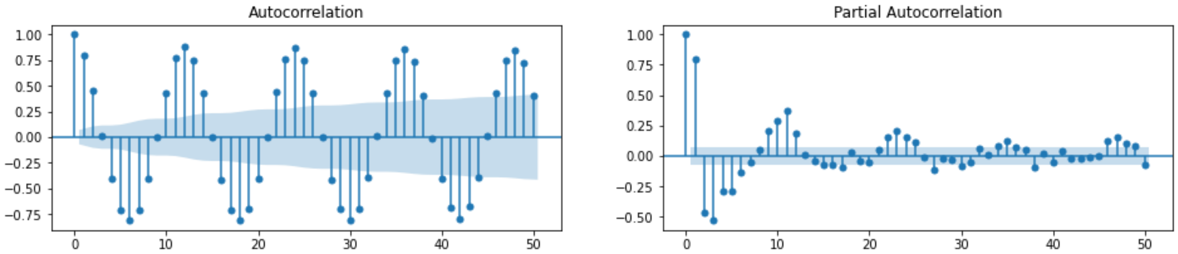
We can use **ACF and PACF** plots to determine the values for **p** and **q**. We must use a stationary time series for ACF and PACF plots. These plots show the relationship between a time series value and its lagged values.

An **autocorrelation** function at lag 5 will measure the relationship between values Y(t) and Y(t-5). That is the correlation of Y(t1), Y(t2), Y(t3) etc. with Y(t1-5), Y(t2-5), Y(t3-5) respectively.

The **partial autocorrelation** between two variables is the amount of correlation that is not explained by the other mutual lags. For example, if we are regressing Y on X1, X2 and X3, then the partial correlation between Y and X3 is the correlation between them that is not explained by their correlation with X1 and X2.

The two plots can be generated using the code below.

```
fig, axes = plt.subplots(1,2,figsize=(16,3))
plot_acf(dataframe.column.tolist(), lags=50, ax=axes[0])
plot_pacf(dataframe.column.tolist(), lags=50, ax=axes[1])
```



AR Term (p)

We look at the PACF plot for the AR term. The lag at which the PACF cuts off is the indicated number of AR terms. The lag value where the PACF chart crosses the upper confidence interval (The blue shaded region) for the first time. In the example above the 0^{th} and 1^{st} lags are positive and above the confidence interval. The 2^{nd} lag drops below the shaded region. Therefore, the p term is taken as 2.

MA Term (q)

For the q term we look at the ACF plot. The lag at which the ACF cuts off is the indicated number of MA terms. The lag value where the ACF chart crosses the upper confidence interval for the first time. In the above example we see that the 3^{rd} lag is where the plot crosses the shaded region. Therefore, q is 3.

For more information on determining the ARIMA parameters check the references [3] and [4] below.

Auto ARIMA and Seasonal ARIMA

Determining the parameters from ACF and PACF plots is time taking and confusing. We can use python packages for doing that automatically. The code below uses **auto_arima** to determine the best parameters and fit the model automatically. Refer section 6 in the ARIMA notebooks for the full code

```python
# Seasonal - fit stepwise auto-ARIMA
smodel = None
smodel = pm.auto_arima(perthTemp_Train_MS[['mintemp']], start_p=1, start_q=1,
                         test='adf',
                         max_p=9, max_q=9, m=12,
                         start_P=0, seasonal=True,
                         d=None, D=1, trace=True,
                         error_action='ignore',
                         suppress_warnings=True,
                         stepwise=True)

smodel.summary()
```

The output from the above code is shown below.

```
 ARIMA(3,0,4)(2,1,0)[12] intercept   : AIC=inf, Time=9.70 sec
 ARIMA(2,0,3)(2,1,0)[12]             : AIC=2665.060, Time=1.22 sec
 ARIMA(2,0,3)(1,1,0)[12]             : AIC=2746.145, Time=0.64 sec
 ARIMA(2,0,3)(2,1,1)[12]             : AIC=inf, Time=5.97 sec
 ARIMA(2,0,3)(1,1,1)[12]             : AIC=inf, Time=2.84 sec
 ARIMA(1,0,3)(2,1,0)[12]             : AIC=2668.655, Time=1.08 sec
 ARIMA(2,0,2)(2,1,0)[12]             : AIC=2666.399, Time=1.09 sec
 ARIMA(3,0,3)(2,1,0)[12]             : AIC=inf, Time=5.42 sec
 ARIMA(2,0,4)(2,1,0)[12]             : AIC=2665.168, Time=1.30 sec
 ARIMA(1,0,2)(2,1,0)[12]             : AIC=2668.620, Time=0.78 sec
 ARIMA(1,0,4)(2,1,0)[12]             : AIC=2665.581, Time=1.29 sec
 ARIMA(3,0,2)(2,1,0)[12]             : AIC=2665.991, Time=1.37 sec
 ARIMA(3,0,4)(2,1,0)[12]             : AIC=inf, Time=5.87 sec

Best model:  ARIMA(2,0,3)(2,1,0)[12]
Total fit time: 142.876 seconds
```

As you can see the Auto ARIMA model picked the best p, q and d parameters to be 2, 0 and 3 like we had predicted before, using the ACF and PACF plots.

This model can now be used to generate forecasts for the future dates. The code is shown below.

```python
n_periods = len(test_dataframe)

fitted, confidence_interval = smodel.predict(n_periods=n_periods,
                                              return_conf_int=True)
```

For more information about Auto ARIMA check the reference [5] below.

5. Multivariate Analysis

ARIMA is a model for univariate analysis. It takes a single time series and forecasts the future values of that time series. **Multivariate time series analysis** is where there are more than 1 columns of data and all the variables are used to forecast the future values of each other.

The notebooks VAR_Multivariate and VAR_NonStationary contain the full code explained below.

Vector Auto Regression (VAR)

Vector Autoregression (VAR) is a multivariate forecasting algorithm that is used when two or more time series influence each other. In the VAR model, each variable is modeled as a linear combination of past values of itself and the past values of other variables in the system.

The code below shows the VAR function from Statsmodels. Check reference [6] below for more info.

```
model = VAR(dataframe)

results = model.fit(ic='aic')

prediction = results.forecast(results.endog, steps =
                                                      len(test_dataframe))
```

The first step before fitting the model is to check the relationship among the variables. We need to check that the variables influence each other. A popular method to check this **is Granger's Causality Test.**

Granger Causality Test

Granger causality test is used to determine if one time series will be useful to forecast another variable by investigating the causality between two variables in a time series.

It is based on the idea that if X causes Y, then the forecast of Y based on previous values of Y and the previous values of X should be better than the forecast of Y based on previous values of Y alone.

It accepts a 2D array with 2 columns as the main argument. The values are in the first column and the predictor (X) is in the second column.

The Null hypothesis assumes that the series in the second column, does not cause the series in the first. If the P-Values are less than a significance level (0.05) then you reject the null hypothesis and conclude that the said lag of X is indeed useful.

The code below shows a function that takes a dataframe of time series values and returns a matrix of p values. Each column of data in the given dataframe is checked for causality with every other column.

```
def grangers_causality_matrix(X_train, variables, test = 'ssr_chi2test', verbose=False):

    dataset = pd.DataFrame(np.zeros((len(variables), len(variables))), columns=variables, index=variables)

    for c in dataset.columns:
        for r in dataset.index:

            test_result = grangercausalitytests(X_train[[r,c]], maxlag=maxlag, verbose=False)
            p_values = [round(test_result[i+1][0][test][1],4) for i in range(maxlag)]
            if verbose: print(f'Y = {r}, X = {c}, P Values = {p_values}')
            max_p_value = np.max(p_values)
            dataset.loc[r,c] = max_p_value

    dataset.columns = [var + '_x' for var in variables]
    dataset.index = [var + '_y' for var in variables]

    return dataset
```

Each variable is tested as both a predictor and target. The below image shows the result. If we look at the value in row 2, column 1 (0.0022), it tells us that the p value for the hypothesis **'mintemp'** not causing **'maxtemp'** is 0.0022. Therefore, we can reject the null hypothesis and conclude that mintemp has an effect on maxtemp. Similarly, since all the p values are below the significance level, we can conclude that all the variables have an effect on each other and move forward with the Vector Auto Regression model.

```
grangers_causality_matrix(perthTemp_Train_MS, variables = perthTemp_Train_MS.columns, verbose = True)
```

```
Y = mintemp, X = mintemp, P Values = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Y = maxtemp, X = mintemp, P Values = [0.0, 0.0, 0.0, 0.0004, 0.0001, 0.0001, 0.0001, 0.0002, 0.0005, 0.0022, 0.0014, 0.0007]
Y = rainfall, X = mintemp, P Values = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Y = mintemp, X = maxtemp, P Values = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Y = maxtemp, X = maxtemp, P Values = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Y = rainfall, X = maxtemp, P Values = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Y = mintemp, X = rainfall, P Values = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Y = maxtemp, X = rainfall, P Values = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0001, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Y = rainfall, X = rainfall, P Values = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

	mintemp_x	maxtemp_x	rainfall_x
mintemp_y	1.0000	0.0	0.0000
maxtemp_y	0.0022	1.0	0.0001
rainfall_y	0.0000	0.0	1.0000

For more information on using Granger Causality check references [7] and [8] below.

Manual Differencing and Inverting

Unlike ARIMA, VAR does not perform a differencing operation by itself. This means that we must check the stationarity of each of the variables independently and then difference those columns that are not stationary.

Since the differencing operation is done manually, after generating the forecast values they must be transformed back into actual values for the forecast. This inverse differencing also needs to be done manually. The process is shown below.

Code for the initial differencing operation is shown below. It is done by a simple pandas method.

```
Differenced_data = data_frame.diff()
```

When forecasts are generated using differenced data the output is also differenced.

```
forecasts = results.forecast(results.endog, steps =
                                            len(perthTemp_Test_MS))
```

The differenced forecast values need to be reverse differenced to get the actual forecast values. The inverse differencing operation is shown below.

```
forecast_actual['column'] = np.r_[data_frame[-1:]['column'][0],
                                   forecasts['column']].cumsum()[1:]
```

For an example you can check the VAR_NonStationary Notebook section 5.2

Invert Differencing Process

np.r_ is a function that just concatenates its arguments and returns an array. In this case we give it two arguments. The first argument is the last value of the train dataset. The second argument is the differenced forecast values from the VAR model. The numpy function returns a concatenated array.

The returned array contains the last training data value followed by the differenced forecasts.

This array is then given as an input to the cumsum() function. This function returns a cumulative sum array. For example, if we give an array like [1,2,3,4,5] to cumsum() we get this array in return [1, 3, 6, 10, 15]. Each value in the returned array is the cumulative sum of the value in that index position and all the values before it. This in effect inverse differences the values we get from the forecast.

Reference [9] below has more information on the **np.r_** and the **cumsum()** function.

6. Prophet (By Facebook)

The code below can be found in the notebooks Prophet_Univariate and Prophet_Multivariate.

Univariate Analysis

To use Prophet for forecasting, first, a Prophet() object is defined. It is fit on the dataset by calling the fit() function and passing the data. The model has a lot of arguments that we can use to specify the type of growth. Seasonality etc. but by default it figures out everything automatically.

The fit() function takes a DataFrame of time series data. **The DataFrame must have a specific format.** The **first column** must have the name **'ds'** and contain the date-times. The **second column** must have the name **'y'** and contain the observations. The code below shows a basic prophet model.

```
prophet_model = fbprophet.Prophet()

prophet_model.fit(dataframe)

forecastDates =
  pd.DataFrame(test_dataframe.index).rename(columns={'Date':'ds'})

forecast_column = prophet_model.predict(forecastDates)

forecast_column_1 = forecast_column[['ds', 'yhat']]
```

For more information on Prophet, check the references [10] and [11] below.

Multivariate (Additional Regressors)

Prophet can only forecast one value for every model. But additional regressors can be added to the model using the **add_regressor** method or function. A column with the regressor value will need to be present in both the fitting and prediction dataframes.

Like in the previous section we create a dataframe with two columns 'ds' and 'y' which are the time stamps and the target variable. For multivariate forecasting we can added more predictor variable columns like shown below.

	ds	mintemp	maxtemp	y
0	1944-06-03	11.0	22.3	0.0
1	1944-06-04	12.2	23.4	0.0
2	1944-06-05	12.0	20.3	2.0
3	1944-06-06	7.4	18.7	3.3

Here the **mintemp** and **maxtemp** columns will be used as additional regressors. The code below shows the process of adding regressors.

```
model = fbprophet.Prophet()

model.add_regressor('mintemp', standardize=True)

model.add_regressor('maxtemp', standardize=True)

model.fit(perthTemp_Train)
```

The remaining steps for forecasting are the same as before. To forecast the 'y' variable we need to provide the values of the regressors for the forecast dates as well. This means that if we are predicting the future values of a target variable (y), we need to first forecast the additional regressors (mintemp and maxtemp) using separate Prophet models and then use those values to forecast the final target variable y.

In the context of our example, we need to first forecast the values for mintemp and maxtemp using a separate model for each and then use those values as regressors for the final prediction.

For more information on adding regressors to the Prophet model check the documentation at reference [12] below.

7. LSTM (Long Short-Term Memory)

The code for this section can be found in the **LSTM_TSA_Template** notebook.

The preprocessing required for this method is slightly different from the previous models. The same initial steps of checking NULLs and handling dates will be needed which can be seen in the sections 2 to 5 in the notebook.

Scaling

Neural Networks work best when all the variables have a similar range of values. The **MinMaxScaler** will transform the data to lie between -1 and 1 but the distribution of data will be preserved.

```
scaler = MinMaxScaler(feature_range=(-1, 1))

scaler = scaler.fit(trainData)

trainScaled = scaler.transform(trainData)

testScaled = scaler.transform(testData)
```

All the prior models took the time series data in the form of a single dataframe where each row corresponds to a single time instance. LSTM is a supervised model that expects data in the form of an input and output. The input will be a certain number of past values from any number of variables and the output will be the next timestamp value of the target variable.

For example, say the table below shows 4 predictor variables and one target variable.

A	b	c	d	t
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30

To train the LSTM to predict the next value of 't' using 2 previous time steps we need to give the input as shown below.

[1 2 3 4][6 7 8 9] → 15

[6 7 8 9][11 12 13 14] → 20

[11 12 13 14][16 17 18 19] → 25

And then when we give the model the input [16 17 18 19][21 22 23 24] it will try to predict the target value of the next time step the true value of which is 30.

The Keras deep learning library has methods that take as input regular timeseries data and transform it into the required format. The code is shown below. The method returns a generator object.

```
TrainGenerator = TimeseriesGenerator(trainX, trainY, length=5)
```

The length argument specifies the number of time steps to go back for each target value. The generator object can be passed as input while training the neural network as shown.

```
history = model.fit(TrainGenerator, epochs=50, validation_data =
TestGenerator, verbose=2)
```

We create another generator for the test data and use it to predict the future values of the target.

For more information check reference [13] and [14] below.

8. Evaluation (Part 3)

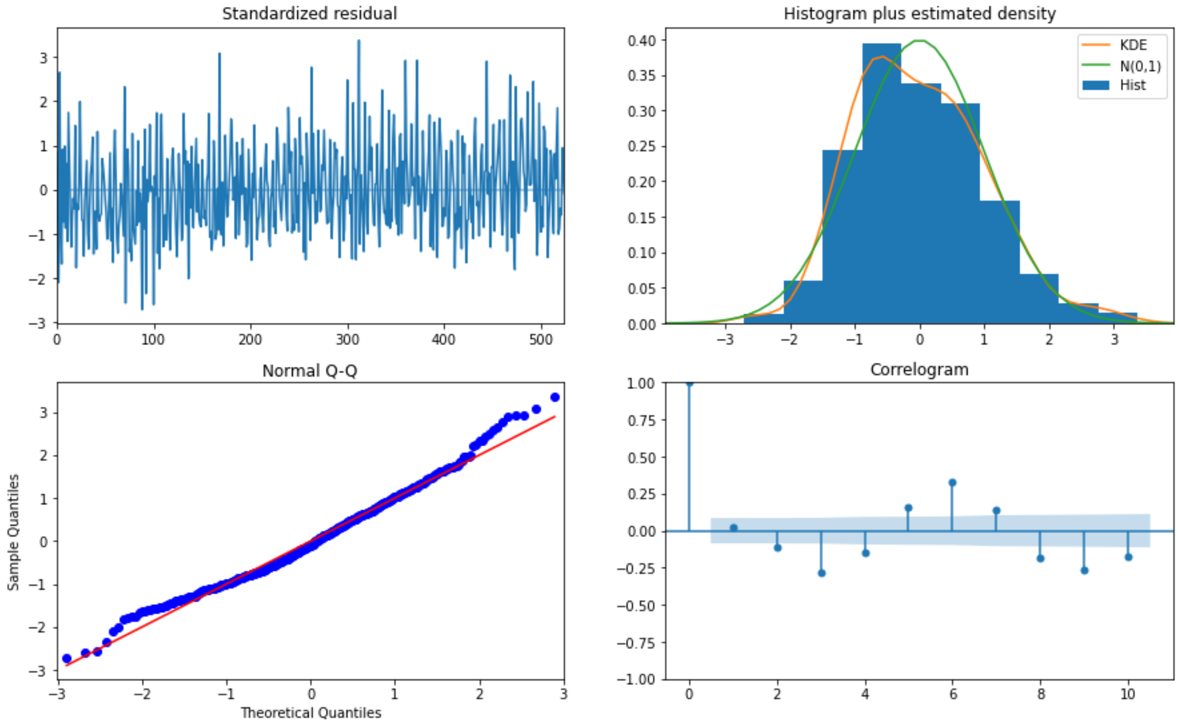
Analyzing Residuals

Residuals in a time series are the differences between the true value and the forecast value. A good forecasting method will give you residuals that are –

- Uncorrelated. That is they are independent and random.
- The residuals have 0 mean.

The distribution of residuals for an ARIMA model can be checked using the code shown below.

```
model.plot_diagnostics(figsize=(15,9))
plt.show()
```



Sci-Kit Learn Metrics

Another common way to evaluate the performance of the forecasts is by calculating some metrics like **Mean Absolute Error, Mean Absolute Percentage Error, Root Mean Square Error** etc. Sci-Kit Learn's regression metrics can be used for this purpose. It offers simple functions to which the true values and the forecasts are provided which them returns the value of the metric. Some functions are shown below.

```
mean_absolute_error(y_true, y_pred)

mean_absolute_percentage_error(y_true, y_pred)

mean_squared_error(y_true, y_pred)
```

Check references [15], [16] and [17] below for more information.

9. References

[1] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html

[2] [3] https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/

[4] http://people.duke.edu/~rnau/411arim3.htm

[5] https://www.analyticsvidhya.com/blog/2018/08/auto-arima-time-series-modeling-python-r/

[6] https://www.machinelearningplus.com/time-series/vector-autoregression-examples-python/

[7] https://towardsdatascience.com/granger-causality-and-vector-auto-regressive-model-for-time-series-forecasting-3226a64889a6

[8] https://www.machinelearningplus.com/time-series/time-series-analysis-python/

[9] https://numpy.org/doc/stable/reference/generated/numpy.r_.html

[9] https://numpy.org/doc/stable/reference/generated/numpy.cumsum.html

[10] https://facebook.github.io/prophet/docs/quick_start.html#python-api

[11] https://machinelearningmastery.com/time-series-forecasting-with-prophet-in-python/#:~:text=The%20Prophet%20library%20is%20an,and%20seasonal%20structure%20by%20default.

[12] https://facebook.github.io/prophet/docs/seasonality,_holiday_effects,_and_regressors.html#additional-regressors

[13] https://machinelearningmastery.com/how-to-use-the-timeseriesgenerator-for-time-series-forecasting-in-keras/

[14] https://machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/

[15] https://machinelearningmastery.com/time-series-forecasting-performance-measures-with-python/#:~:text=The%20mean%20absolute%20error%2C%20or,is%20called%20making%20them%20absolute.

[16] https://joydeep31415.medium.com/common-metrics-for-time-series-analysis-f3ca4b29fe42

[17] https://otexts.com/fpp2/residuals.html