AGENDA

1. Time series Analysis ARIMA ARMA models, Prophet and Neural prophet
2. Feature Engineering for Time Series

3. Parameters Tuning for Time Series

Data Science Academy

I

Time series Analysis

ARIMA ARMA models, Prophet and Neural prophet

Data Science Academy

TIME SERIES

- A time series is a sequential set of data points, measured typically over successive times.
- Time series analysis comprises methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data.

WHY USE TIME SERIES DATA?

- To develop forecasting models
  - What will the rate of inflation be next year?
- To estimate dynamic causal effects
  - If the Fed increases the Federal Funds rate now, what will be the effect on the rates of inflation and unemployment in 3 months? in 12 months?
  - What is the effect over time on cigarette consumption of a hike in the cigarette tax?
- Or, because that is your only option…
  - Rates of inflation and unemployment in the US can be observed only over time!

CATEGORIES AND TERMINOLOGIES

- Time-domain vs. Frequency-domain
  - Time-domain approach: how does what happened today affect what will happen tomorrow? These approaches view the investigation of lagged relationships as most important, e.g. autocorrelation analysis.
  - Frequency-domain approach: what is the economic cycle through periods of expansion and recession? These approaches view the investigation of cycles as most important, e.g. spectral analysis and wavelet analysis
- This lecture will focus on time-domain approaches.

CATEGORIES AND TERMINOLOGIES

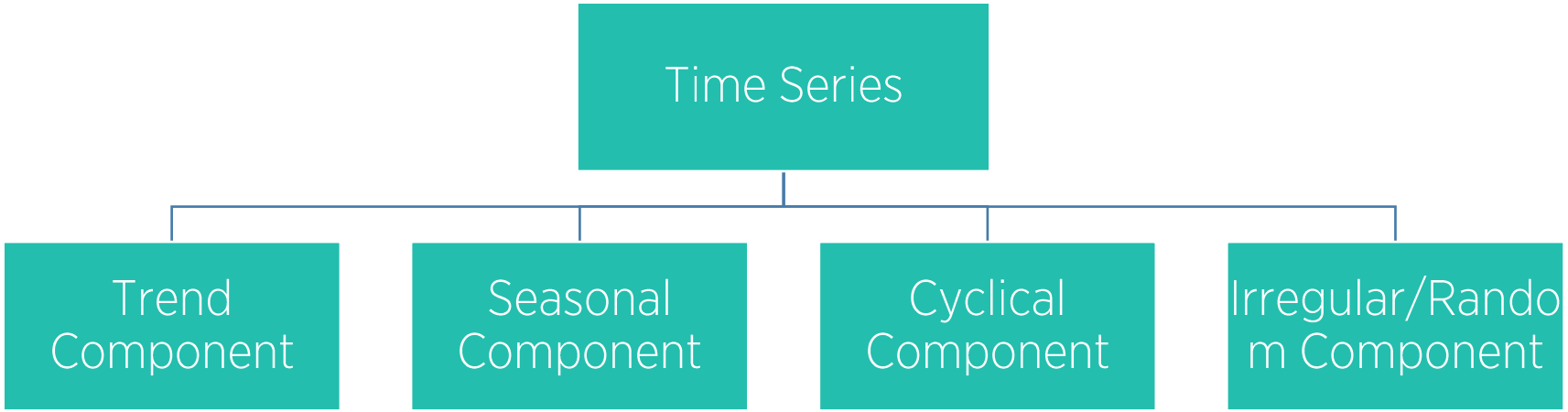
- univariate vs multivariate

  A time series containing records of a single variable is termed as univariate, but if records of more than one variable are considered then it is termed as multivariate
- linear vs. non-linear

  A time series model is said to be linear or non-linear depending on whether the current value of the series is a linear or non-linear function of past observations.
- discrete vs. continuous

  In a continuous time series observations are measured at every instance of time, whereas a discrete time series contains observations measured at discrete points in time.

TIME-SERIES COMPONENTS



Overall, persistent, long-term movement

Regular periodic fluctuations, usually within a 12-month period

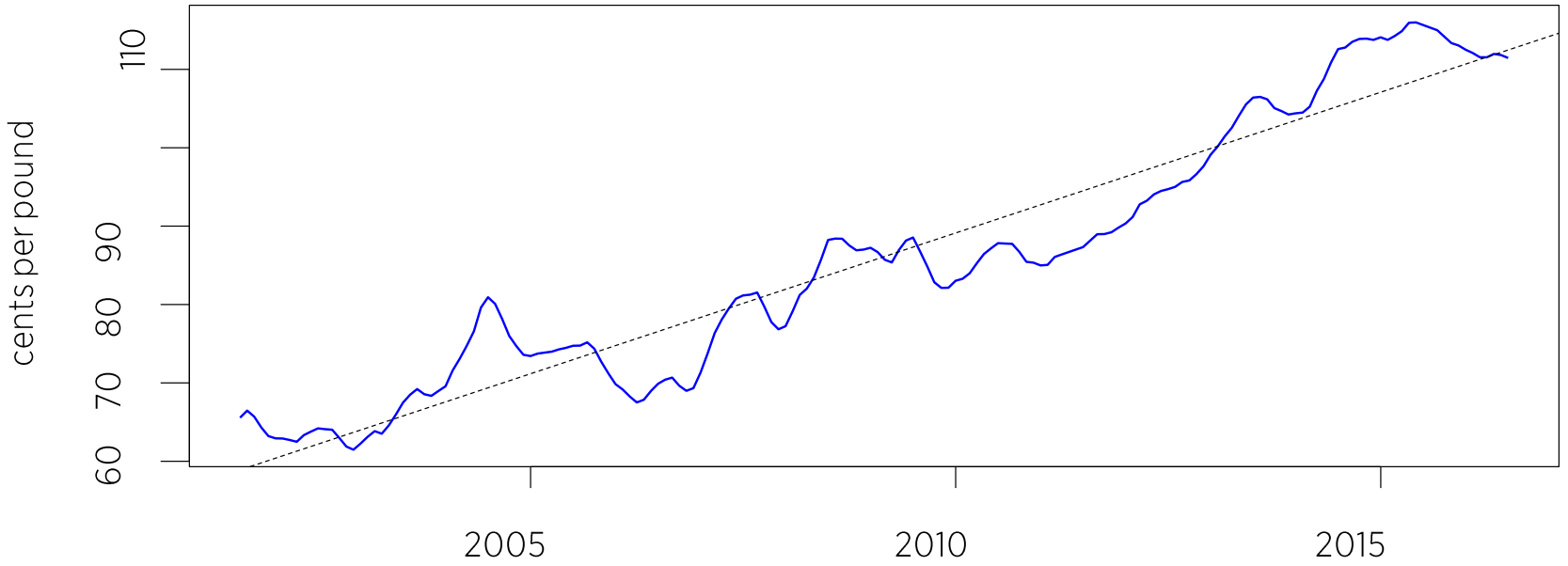
Repeating swings or movements over more than one year

Erratic or residual fluctuations

COMPONENTS OF A TIME SERIES (TREND)

- In general, a time series is affected by four components, i.e. trend, seasonal, cyclical and irregular components.
  - Trend

    The general tendency of a time series to increase, decrease or stagnate over a long period of time.

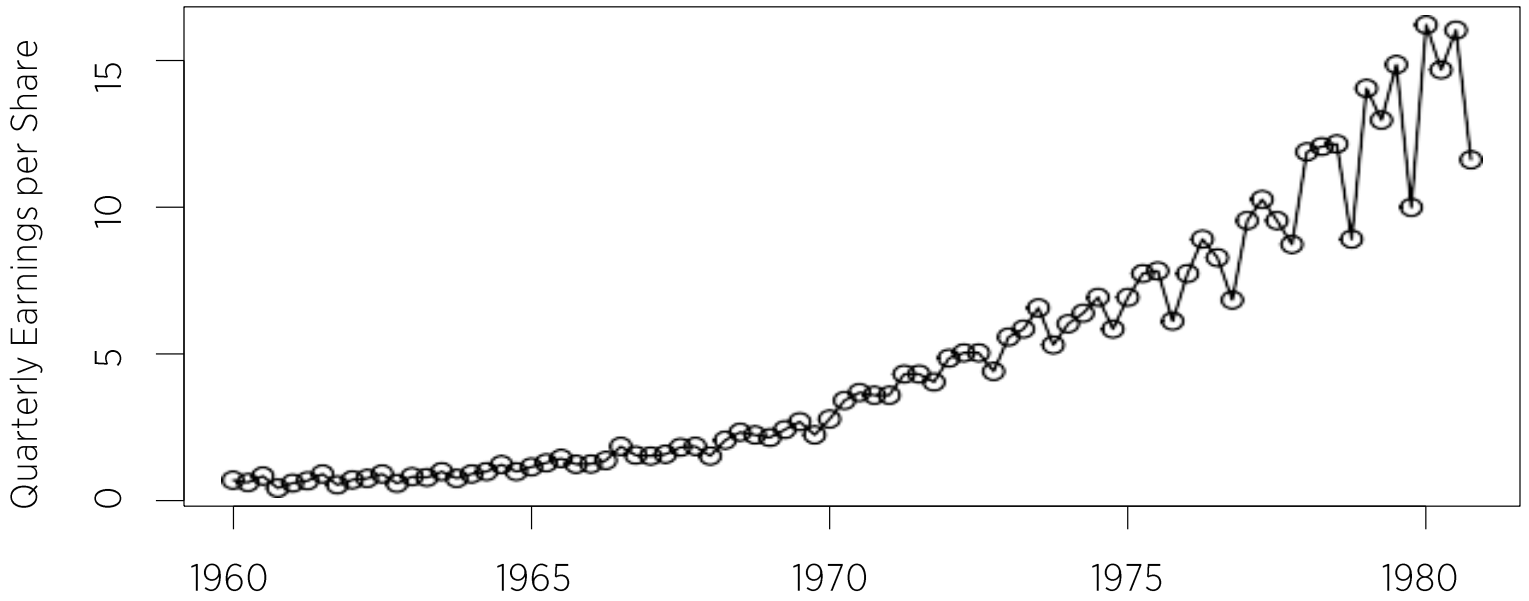


The price of chicken: monthly whole bird spot price, Georgia docks, US cents per pound, August 2001 to July 2016, with fitted linear trend line

COMPONENTS OF A TIME SERIES (Seasonality)

- In general, a time series is affected by four components, i.e. trend, seasonal, cyclical and irregular components.
  - Seasonal variation

    This component explains fluctuations within a year during the season, usually caused by climate and weather conditions, customs, traditional habits, etc.

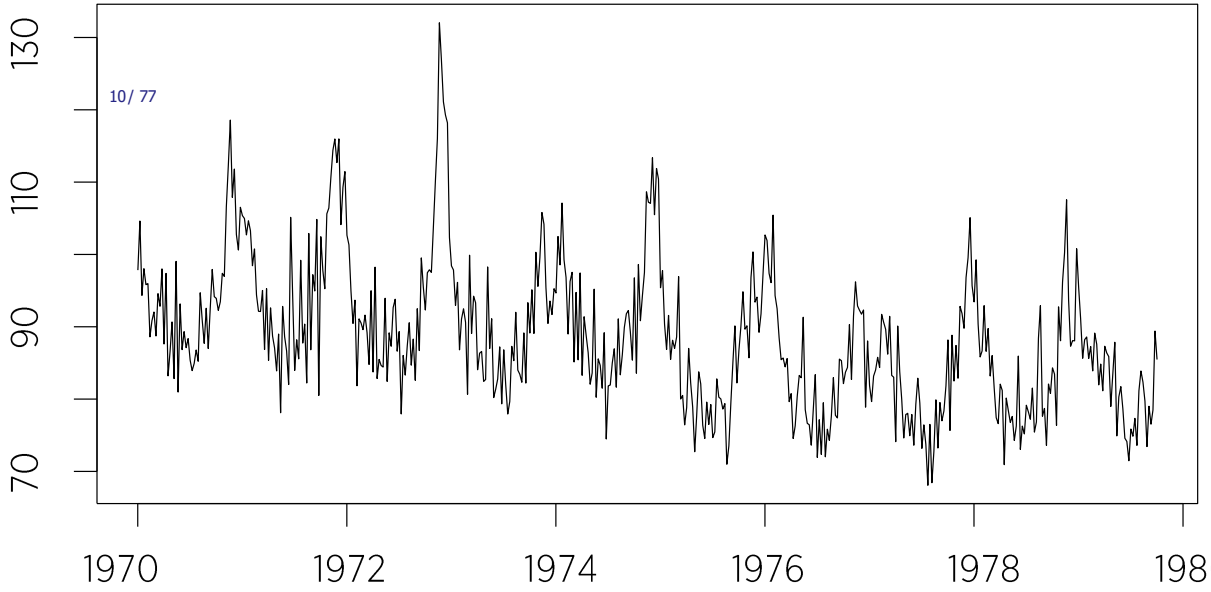


Johnson & Johnson quarterly earnings per share, 84 quarters, 1960-I to 1980-IV.

COMPONENTS OF A TIME SERIES (Cyclic variation)

- In general, a time series is affected by four components, i.e. trend, seasonal, cyclical and irregular components.
  - Cyclical variation

    This component describes the medium-term changes caused by circumstances, which repeat in cycles. The duration of a cycle extends over longer period of time.



Average weekly cardiovascular mortality in Los Angeles County. There are 508 six-day smoothed averages obtained by filtering daily values over the 10 year period 1970-1979.

COMPONENTS OF A TIME SERIES (Irregular variation)

- In general, a time series is affected by four components, i.e. trend, seasonal, cyclical and irregular components
  - Irregular variation

    Irregular or random variations in a time series are caused by unpredictable influences, which are not regular and also do not repeat in a particular pattern.
    These variations are caused by incidences such as war, strike, earthquake, flood, revolution, etc.
    There is no defined statistical technique for measuring random fluctuations in a time series.

Time Series Models

- For analysis the time series data, we have the following model:
  - Additive Model
  - Multiplicative Model
  - Mixed Model

Decomposition of Time Series

- Decomposition is primarily used for time series analysis, and as an analysis tool it can be used to inform forecasting models on your problem.
- It provides a structured way of thinking about a time series forecasting problem, both generally in terms of modeling complexity and specifically in terms of how to best capture each of these components in a given model.
- Real-world problems are messy and noisy. There may be additive and multiplicative components. There may be an increasing trend followed by a decreasing trend. There may be non-repeating cycles mixed in with the repeating seasonality components.
- Nevertheless, these abstract models provide a simple framework that you can use to analyze your data and explore ways to think about and forecast your problem.

Decomposition of Time Series

A given time series is thought to consist of three systematic components including level, trend, seasonality, and one non-systematic component called noise.

These components are defined as follows:

- Level: The average value in the series.
- Trend: The increasing or decreasing value in the series.
- Seasonality: The repeating short-term cycle in the series.
- Noise: The random variation in the series.

ADDITIVE MODEL

- We may set the additive model as,

$$Y_t = T_t + S_t + C_t + R_t$$

- Where,

T_t - Trend
S_t - Seasonal variation
C_t - Cyclic variation
R_t - Irregular variation

MULTIPLICATIVE MODEL

- We may write the multiplicative model as,

$$\boldsymbol{Y_t = T_t \times S_t \times C_t \times R_t}$$

- Where,

T_t - Trend
S_t - Seasonal variation
C_t - Cyclic variation
R_t - Irregular variation

MIXED MODEL

- The mixed model can be written as

$$Y_t = T_t \times C_t + S_t \times R_t$$

$$Y_t = T_t + S_t \times C_t \,.\, R_t$$

- Where,

T_t - Trend
S_t - Seasonal variation
C_t - Cyclic variation
R_t - Irregular variation

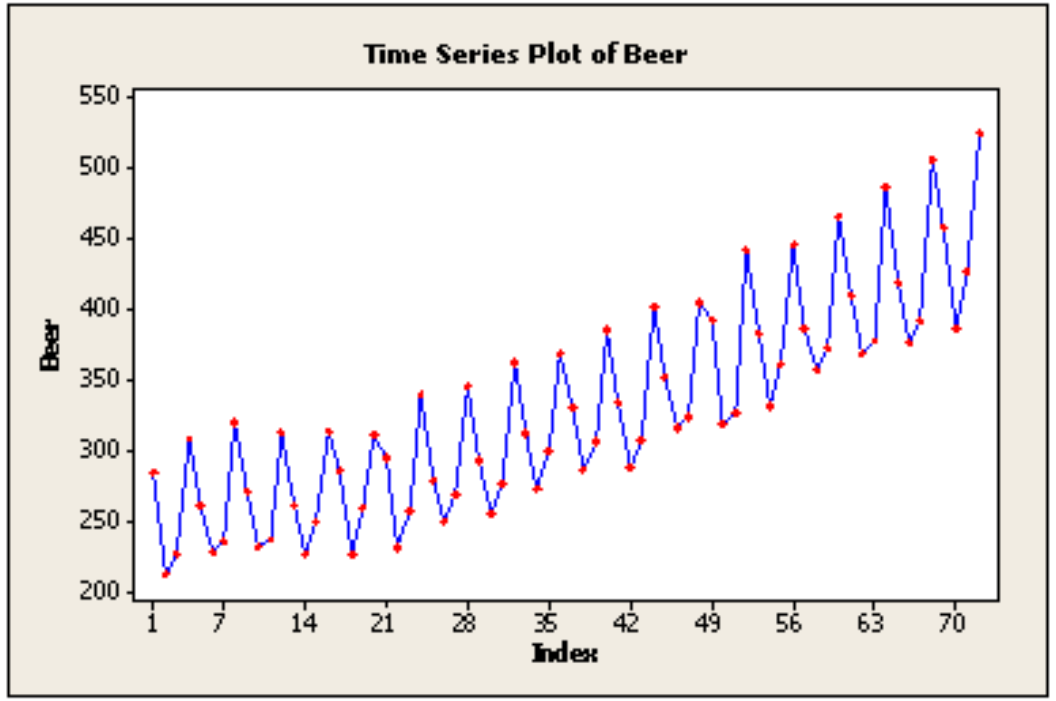
ADDITIVE AND MULTIPLICATIVE. WHAT MODEL TO USE?

- In many time series, the amplitude of both seasonal and irregular variations increase as the level of the trend rises. In this situation, a multiplicative model is usually appropriate.

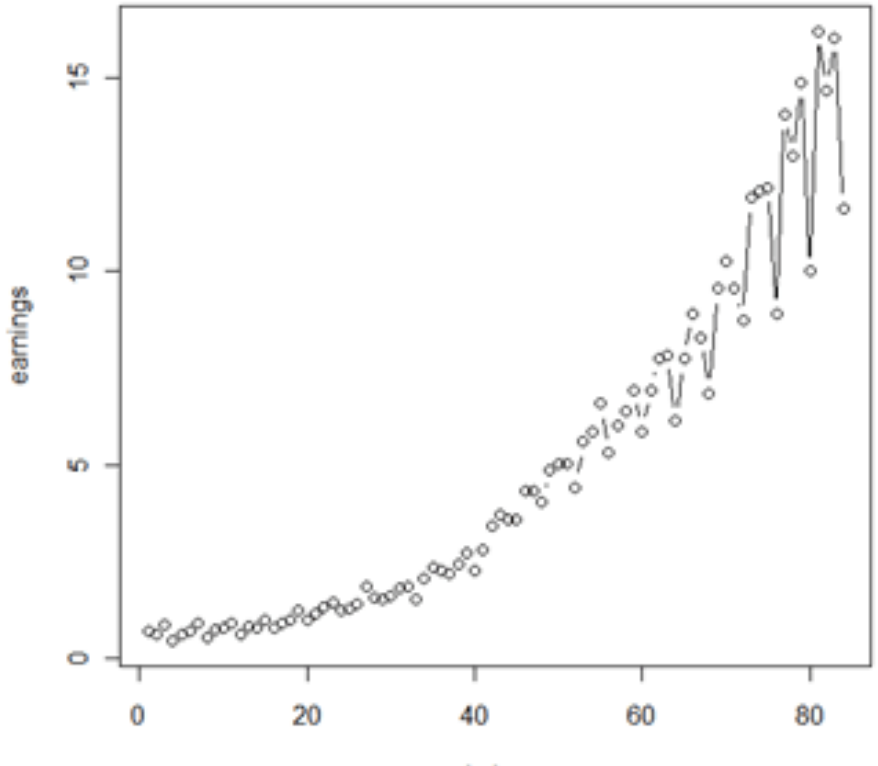
- In many time series, the amplitude of both seasonal and irregular variations do not change as the level of the trend rises or falls. In such cases, an additive model is appropriate.

Additive and Multiplicative. What model to use?

- The given graph of time series shows quarterly beer production in Australia. The seasonal variation looked to be about the same magnitude across time, so an additive decomposition might be good.

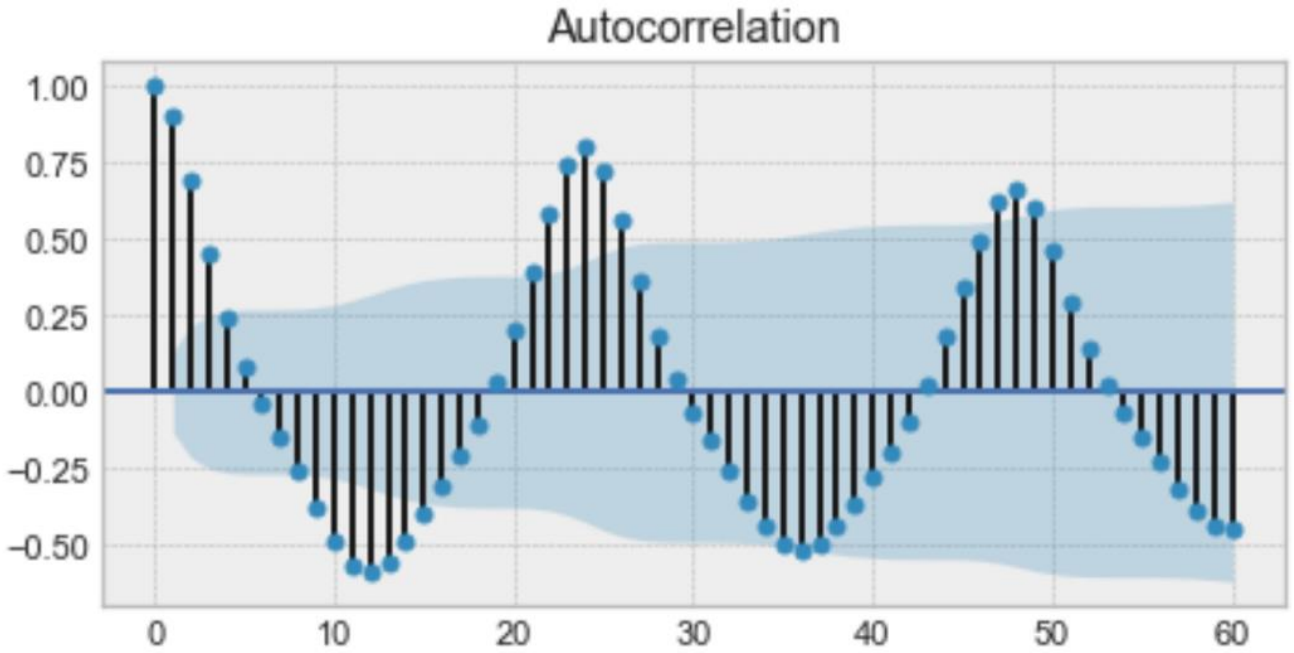


- The quarterly earnings data for the Johnson and Johnson Corporations. The seasonal variation increases as we move across time. A multiplicative decomposition could be useful.



AUTOCORRELLATION

- Autocorrelation is the similarity between observations as a function of the time lag between them.



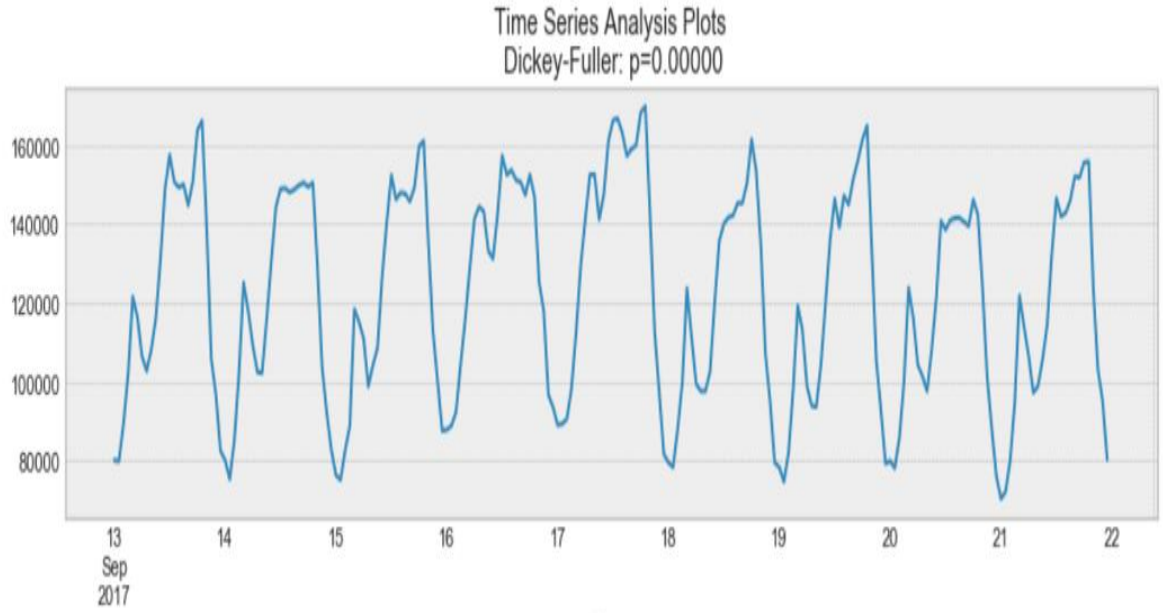
EXAMPLE OF AUTOCORRELATION

- Let’s assume Emma is looking to determine if a stock's returns in her portfolio exhibit autocorrelation; the stock's returns relate to its returns in previous trading sessions. If the returns do exhibit autocorrelation, Emma could characterize it as a momentum stock because past returns seem to influence future returns. Emma runs a regression with two prior trading sessions' returns as the independent variables and the current return as the dependent variable.
- She finds that returns one day prior have a positive autocorrelation of 0.7, while the returns two days prior have a positive autocorrelation of 0.3. Past returns seem to influence future returns. Therefore Emma can adjust her portfolio to take advantage of the autocorrelation and resulting momentum by continuing to hold her position or accumulating more shares.

STATIONARITY

- **Stationarity** is an important characteristic of time series. A time series is said to be stationary if its statistical properties do not change over time. In other words, it has constant mean and variance, and covariance is independent of time.

- Ideally, we want to have a stationary time series for modelling. Of course, not all of them are stationary, but we can make different transformations to make them stationary.



ARIMA MODELS

- Autoregressive (AR) process:
  - Series current values depend on its own previous values
- Moving average (MA) process:
  - The current deviation from mean depends on previous deviations
- Autoregressive Moving average (ARMA) process
- Autoregressive Integrated Moving average (ARIMA)process.
- ARIMA is also known as Box-Jenkins approach. It is popular because of its generality;
- It can handle any series, with or without seasonal elements, and it has well-
- documented computer programs

ARIMA FROM SCRATCH

- Part 1 : Making the data stationary

So many transformations exist for making the data stationary. We used log here and differencing. Subtracting the previous value didn't make it stationary as it had a seasonal trend. Hence, we shifted the data by 12 and then subtracted it again.

```
df_testing = pd.DataFrame(np.log(df.Value).diff().diff(12))
```

ARIMA FROM SCRATCH

- Part 2 : Auto-Regressive Model

  For a value at time t, we assume that it is linearly dependent on the previous p lagged values and there is an error term associated with it.

$$y_t = c + \sum_{i=1}^{p} \varphi_i y_{t-i} + \varepsilon_t = c + \varphi_1 y_{t-1} + \varphi_2 y_{t-2} + \ldots\ldots\ldots\ldots + \varphi_p y_{t-p} + \varepsilon_t$$

  It is similar to linear regression, where X is the p -lagged values and y is the value at time t. Here the order **p** can be chosen using ACF and PACF plots, or if you have enough data just think of it as a hyper-parameter.

ARIMA FROM SCRATCH

- Autoregressive model

```
def AR(p,df):
  df_temp = df

  #Generating the lagged p terms
  for i in range(1,p+1):
    df_temp['Shifted_values_%d' % i ] =
df_temp['Value'].shift(i)

  train_size = (int)(0.8 * df_temp.shape[0])

  #Breaking data set into test and training
  df_train = pd.DataFrame(df_temp[0:train_size])
  df_test = pd.DataFrame(df_temp[train_size:df.shape[0]])

  df_train_2 = df_train.dropna()
  #X contains the lagged values ,hence we skip the first column
  X_train = df_train_2.iloc[:,1:].values.reshape(-1,p)
  #Y contains the value,it is the first column
  y_train = df_train_2.iloc[:,0].values.reshape(-1,1)
```

```
#Running linear regression to generate the coefficents of lagged terms
  from sklearn.linear_model import LinearRegression
  lr = LinearRegression()
  lr.fit(X_train,y_train)

  theta  = lr.coef_.T
  intercept = lr.intercept_
  df_train_2['Predicted_Values'] = X_train.dot(lr.coef_.T) + lr.intercept_
  # df_train_2[['Value','Predicted_Values']].plot()

  X_test = df_test.iloc[:,1:].values.reshape(-1,p)
  df_test['Predicted_Values'] = X_test.dot(lr.coef_.T) + lr.intercept_
  # df_test[['Value','Predicted_Values']].plot()

  RMSE = np.sqrt(mean_squared_error(df_test['Value'],
df_test['Predicted_Values']))

  print("The RMSE is :", RMSE,", Value of p : ",p)
  return [df_train_2,df_test,theta,intercept,RMSE]
```

ARIMA FROM SCRATCH

- Part 3 : Moving Average

Similar to the p, q is used here to denote the number of lagged observations. Again it can be thought of a linear regression with q-lagged errors as X and the error as y.

```
def MA(q,res):

  for i in range(1,q+1):
    res['Shifted_values_%d' % i ] = res['Residuals'].shift(i)

  train_size = (int)(0.8 * res.shape[0])

  res_train = pd.DataFrame(res[0:train_size])
  res_test = pd.DataFrame(res[train_size:res.shape[0]])

  res_train_2 = res_train.dropna()
  X_train = res_train_2.iloc[:,1:].values.reshape(-1,q)
  y_train = res_train_2.iloc[:,0].values.reshape(-1,1)


  print("The RMSE is :", RMSE,", Value of q : ",q)
  return [res_train_2,res_test,theta,intercept,RMSE]
```

```
from sklearn.linear_model import LinearRegression
  lr = LinearRegression()
  lr.fit(X_train,y_train)

  theta  = lr.coef_.T
  intercept = lr.intercept_
  res_train_2['Predicted_Values'] = X_train.dot(lr.coef_.T) +
lr.intercept_
  # res_train_2[['Residuals','Predicted_Values']].plot()

  X_test = res_test.iloc[:,1:].values.reshape(-1,q)
  res_test['Predicted_Values'] = X_test.dot(lr.coef_.T) + lr.intercept_
  res_test[['Residuals','Predicted_Values']].plot()

from sklearn.metrics import mean_squared_error
  RMSE = np.sqrt(mean_squared_error(res_test['Residuals'],
res_test['Predicted_Values']))
```

ARIMA FROM SCRATCH

- We now make combine the two AR and MA model to get our predictions

$$ARMA(p,q) : Y_t = c + \sum_{i=1}^{p} \phi_i Y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t$$

- Part 4 : Un-differencing the data

```
df_c.Value += np.log(df).shift(1).Value
df_c.Value += np.log(df).diff().shift(12).Value
df_c.Predicted_Values += np.log(df).shift(1).Value
df_c.Predicted_Values += np.log(df).diff().shift(12).Value
df_c.Value = np.exp(df_c.Value)
df_c.Predicted_Values = np.exp(df_c.Predicted_Values)
```

FACEBOOK PROPHET

- "Prophet" is an open-sourced library available on R or Python which helps users analyze and forecast time-series values released in 2017. With developers' great efforts to make the time-series data analysis be available without expert works, it is highly user-friendly but still highly customizable, even to non-expert users.

- Prophet can handle;
  - trend with its changepoints,
  - seasonality (yearly, weekly, daily, and other user-defined seasonality),
  - holiday effect, and
  - input regressors

FACEBOOK PROPHET

- You may find everything is prepared to be user-friendly without any special care about the time-series data handling. Once you are familiar with basic Python data modeling using sklearn APIs, Prophet code should also look similar.

```
import pandas as pd
from prophet import Prophet
df = pd.read_csv('../examples/example_wp_log_peyton_manning.csv')
```

- We fit the model by instantiating a new Prophet object. Any settings to the forecasting procedure are passed into the constructor. Then you call its fit method and pass in the historical dataframe. Fitting should take 1-5 seconds.

```
m = Prophet()
m.fit(df)
```

PREDICTION WITH FB PROPHET

- Predictions are then made on a dataframe with a column ds containing the dates for which a prediction is to be made. You can get a suitable dataframe that extends into the future a specified number of days using the helper method Prophet.make_future_dataframe.

	DS
3265	2017-01-15
3266	2017-01-16
3267	2017-01-17
3268	2017-01-18
3269	2017-01-19

```
future = m.make_future_dataframe(periods=365)
future.tail()
```

PREDICTION WITH FB Prophet

- The predict method will assign each row in future a predicted value which it names yhat. If you pass in historical dates, it will provide an in-sample fit. The forecast object here is a new dataframe that includes a column yhat with the forecast, as well as columns for components and uncertainty intervals

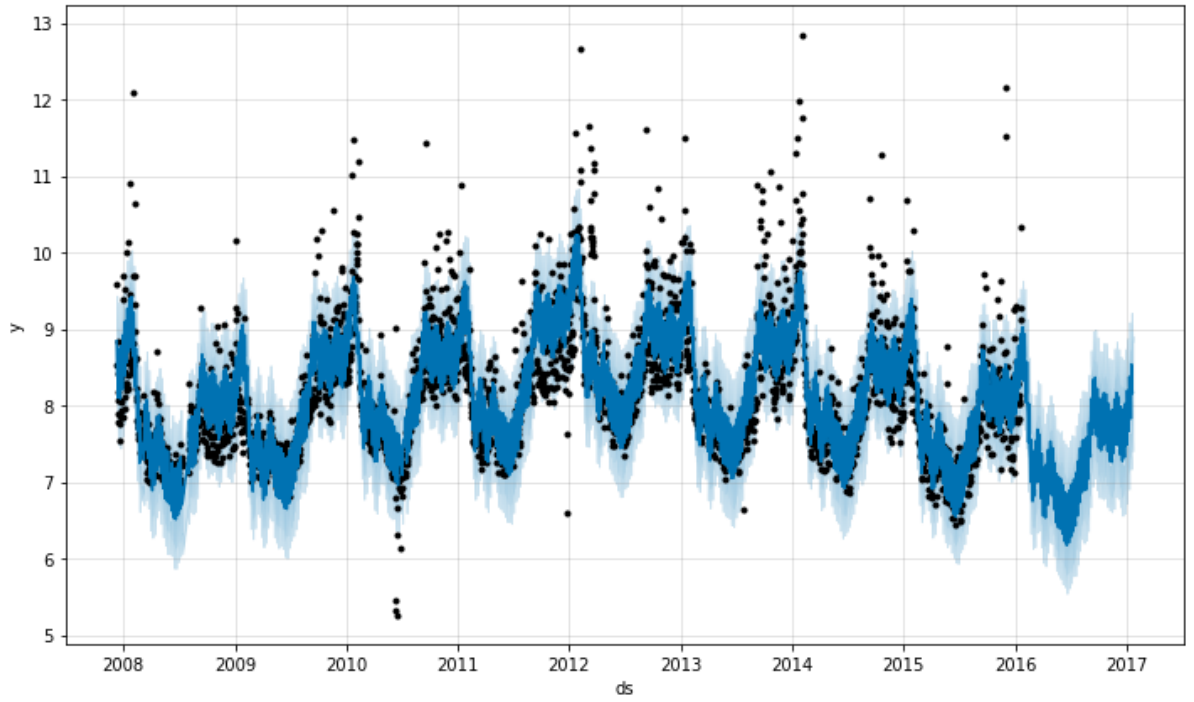
```
forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
```

	DS	YHAT	YHAT_LOWER	YHAT_UPPER
3265	2017-01-15	8.211542	7.444742	8.903545
3266	2017-01-16	8.536553	7.847804	9.21115
3267	2017-01-17	8.323968	7.541829	9.035461
3268	2017-01-18	8.156621	7.404457	8.830642
3269	2017-01-19	8.168561	7.438865	8.908668

PLOTTING RESULTS

- You can plot the forecast by calling the Prophet.plot method and passing in your forecast dataframe.

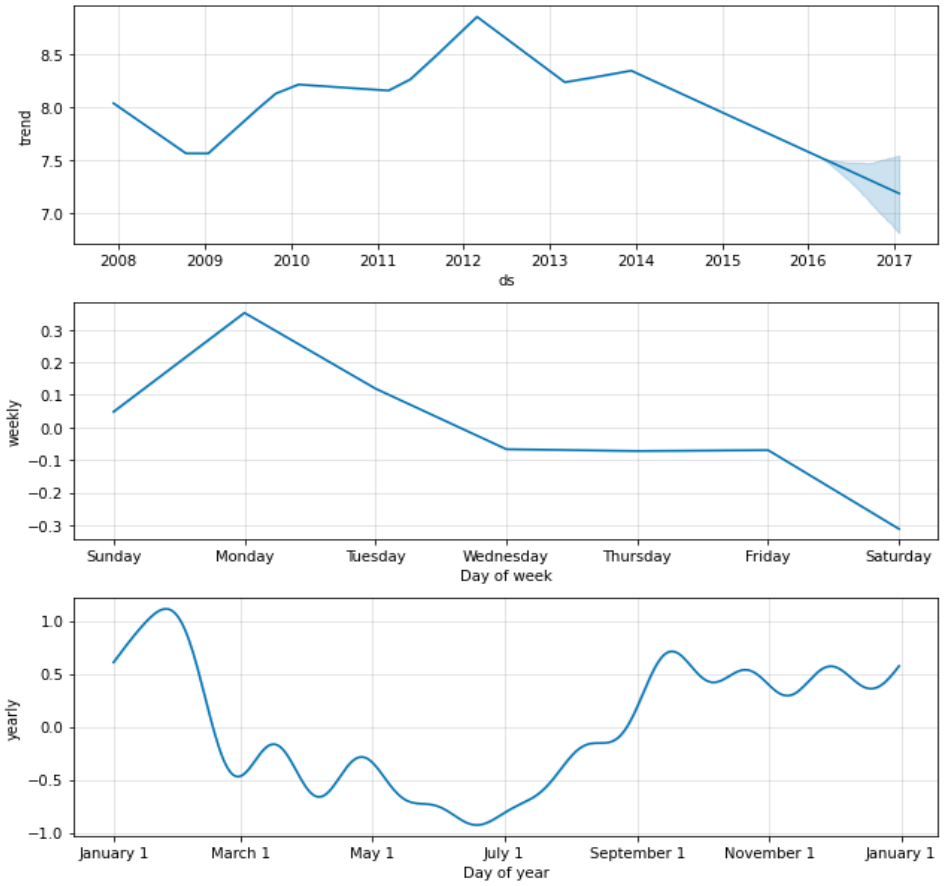
```
fig1 = m.plot(forecast)
```



PLOTTING RESULTS

- If you want to see the forecast components, you can use the Prophet.plot_ components method. By default you'll see the trend, yearly seasonality, and weekly seasonality of the time series

```
fig2 = m.plot_components(forecast)
```



NEURAL PROPHET

- Neural Prophet is a python library for modeling time-series data based on neural networks. It's built on top of PyTorch and is heavily inspired by Facebook Prophet and AR-Net libraries.

Neural Prophet vs. Prophet

- Using PyTorch's Gradient Descent optimization engine making the modeling process much faster than Prophet
- Using AR-Net for modeling time-series autocorrelation
- Custom losses and metrics
- Having configurable non-linear layers of feed-forward neural networks

SIMPLE MODEL WITH NEURAL PROPHET

- A simple model with neural_prophet for this dataset can be fitted by creating an object of the NeuralProphet class as follows and calling the fit function. This fits a model with the default settings in the model.

```
from neuralprophet import NeuralProphet
import pandas as pd
df = pd.read_csv('../example_data/example_wp_log_peyton_manning.csv')
m = NeuralProphet()
metrics = m.fit(df, freq="D")
```

- Once the model is fitted, we can make forecasts using the fitted model. For this, we first need to create a future dataframe consisting of the time steps into the future that we need to forecast for. NeuralProphet provides the helper function make_future_dataframe for this purpose.

```
future = m.make_future_dataframe(df, periods=365)
forecast = m.predict(future)
```

II

Feature Engineering for Time Series

Data Science Academy

DATE-RELATED FEATURES

- Having information about the day, month, year, etc. can be useful for forecasting the value. Extracting these features is really easy in Python.

```
import pandas as pd
data = pd.read_csv('Train_SU63ISt.csv')
data['Datetime'] =
pd.to_datetime(data['Datetime'],format='%d-%m-%Y %H:%M')

data['year']=data['Datetime'].dt.year
data['month']=data['Datetime'].dt.month
data['day']=data['Datetime'].dt.day

data['dayofweek_num']=data['Datetime'].dt.dayofweek
data['dayofweek_name']=data['Datetime'].dt.weekday_name

data.head()
```

TIME-BASED FEATURES

- We can similarly extract more granular features if we have the time stamp. For instance, we can determine the hour or minute of the day when the data was recorded and compare the trends between the business hours and non-business hours.
- Extracting time-based features is very similar to what we did above when extracting date-related features. We start by converting the column to DateTime format and use the .dt accessor.

```
import pandas as pd
data = pd.read_csv('Train_SU63ISt.csv')
data['Datetime'] = pd.to_datetime(data['Datetime'],format='%d-%m-%Y %H:%M')

data['Hour'] = data['Datetime'].dt.hour
data['minute'] = data['Datetime'].dt.minute

data.head()
```

LAG FEATURES

- Here's something most aspiring data scientists don't think about when working on a time series problem – we can also use the target variable for feature engineering!
- Consider this – you are predicting the stock price for a company. So, the previous day's stock price is important to make a prediction, right? In other words, the value at time t is greatly affected by the value at time t-1. The past values are known as lags, so t-1 is lag 1, t-2 is lag 2, and so on.

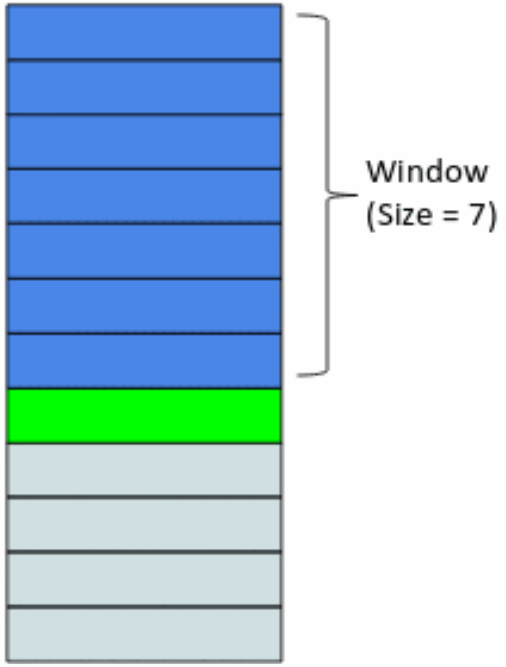
```
import pandas as pd
data = pd.read_csv('Train_SU63ISt.csv')
data['Datetime'] =
pd.to_datetime(data['Datetime'],format='%d-%m-%Y
%H:%M')

data['lag_1'] = data['Count'].shift(1)
data = data[['Datetime', 'lag_1', 'Count']]
data.head()
```

	Datetime	lag_1	Count
0	2012-08-25 00:00:00	NaN	8
1	2012-08-25 01:00:00	8.0	2
2	2012-08-25 02:00:00	2.0	6
3	2012-08-25 03:00:00	6.0	2
4	2012-08-25 04:00:00	2.0	2

ROLLING WINDOW FEATURE

- This method is called the rolling window method because the window would be different for every data point. We will select a window size, take the average of the values in the window, and use it as a feature.

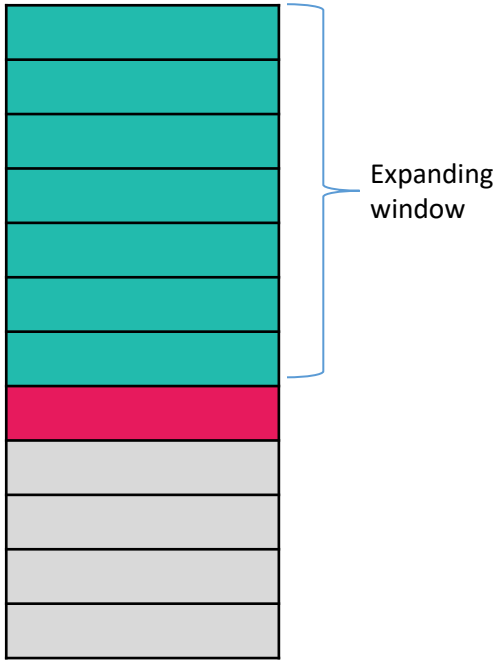
```
import pandas as pd
data = pd.read_csv('Train_SU63ISt.csv')
data['Datetime'] =
pd.to_datetime(data['Datetime'],format='%d-%m-%Y
%H:%M')

data['rolling_mean'] =
data['Count'].rolling(window=7).mean()
data = data[['Datetime', 'rolling_mean', 'Count']]
data.head(10)
```



EXPANDING WINDOW FEATURE

- This is simply an advanced version of the rolling window technique. In the case of a rolling window, the size of the window is constant while the window slides as we move forward in time. Hence, we consider only the most recent values and ignore the past values. The idea behind the expanding window feature is that it takes all the past values into account.

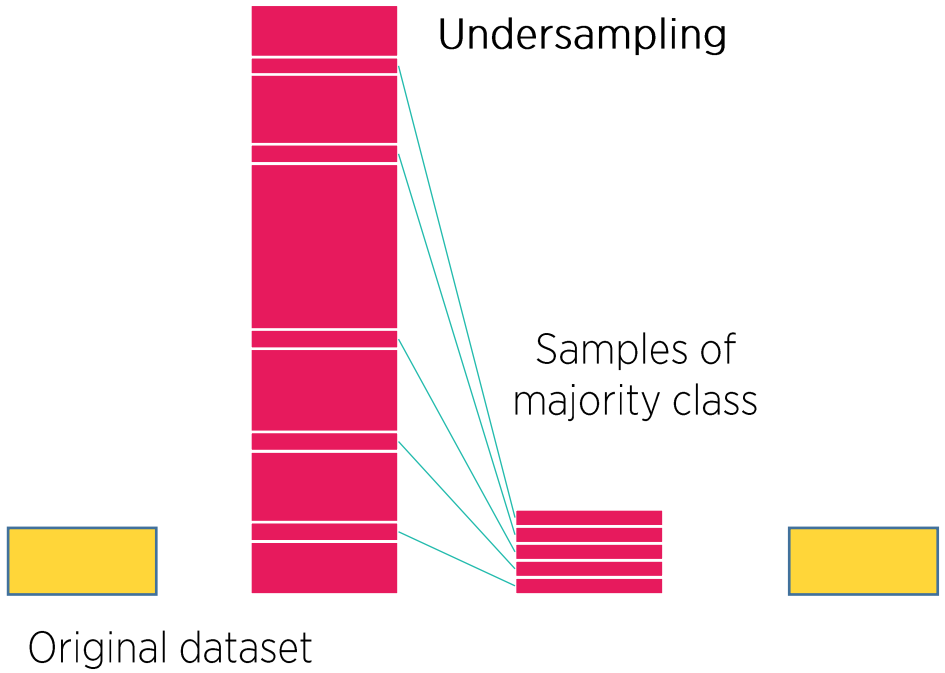
```
import pandas as pd
data = pd.read_csv('Train_SU63ISt.csv')
data['Datetime'] =
pd.to_datetime(data['Datetime'],format
='%d-%m-%Y %H:%M')

data['expanding_mean'] =
data['Count'].expanding(2).mean()
data = data[['Datetime','Count',
'expanding_mean']]
data.head(10)
```

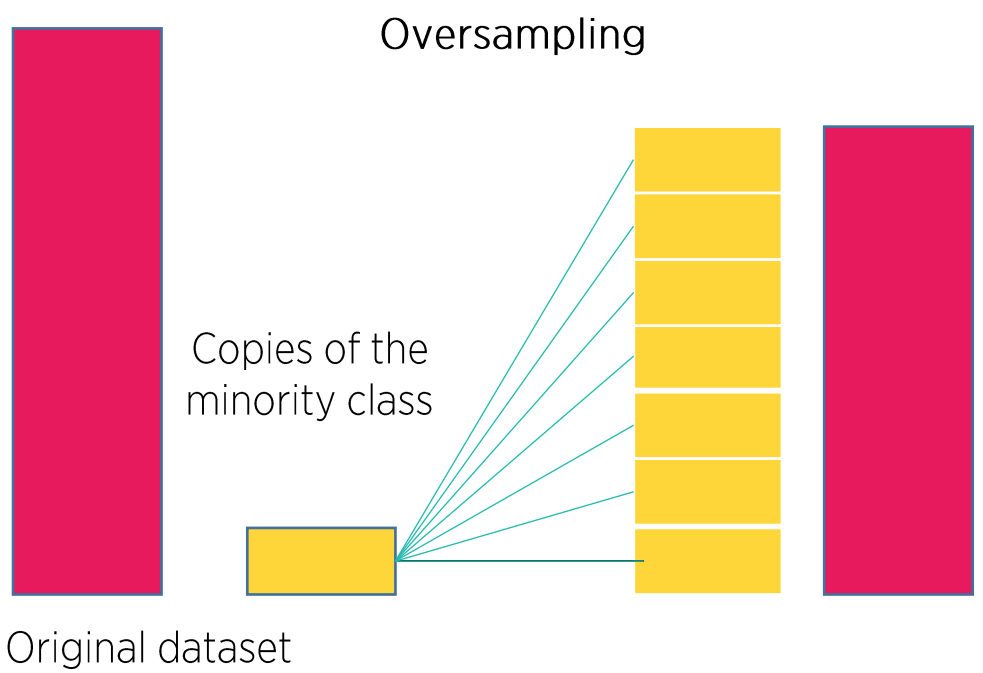


	Datetime	Count	expanding_mean
0	2012-08-25 00:00:00	8	NaN
1	2012-08-25 01:00:00	2	5.000000
2	2012-08-25 02:00:00	6	5.333333
3	2012-08-25 03:00:00	2	4.500000
4	2012-08-25 04:00:00	2	4.000000
5	2012-08-25 05:00:00	2	3.666667
6	2012-08-25 06:00:00	2	3.428571
7	2012-08-25 07:00:00	2	3.250000
8	2012-08-25 08:00:00	6	3.555556
9	2012-08-25 09:00:00	2	3.400000

RESAMPLING

- Often you need to summarize or aggregate time series data by a new time period. For instance, you may want to summarize hourly data to provide a daily maximum value.
- This process of changing the time period that data are summarized for is often called resampling.
- Lucky for you, there is a nice resample() method for pandas dataframes that have a datetime index.





DOWNSAMPLING AND PERFORMING AGGREGATION

- Downsampling is to resample a time-series dataset to a wider time frame. For example, from minutes to hours, from days to years. The result will have a reduced number of rows and values can be aggregated with mean(), min(), max(), sum() etc .

Date	num_sold
2017-01-02 09:02:03	5
2017-01-02 09:14:13	7
2017-01-02 09:21:00	5
2017-01-02 09:28:57	9
2017-01-02 09:42:14	1
...	...
2017-01-02 22:46:36	5

```
df_sales.resample('2H').sum()
```

Date	num_sold
2017-01-02 08:00:00	36
2017-01-02 10:00:00	66
2017-01-02 12:00:00	81
2017-01-02 14:00:00	50
2017-01-02 16:00:00	64
2017-01-02 18:00:00	66
2017-01-02 20:00:00	44
2017-01-02 22:00:00	45

UPSAMPLING AND FILLING VALUES

- Upsampling is the opposite operation of downsampling. It resamples a time-series dataset to a smaller time frame. For example, from hours to minutes, from years to days. The result will have an increased number of rows and additional rows values are defaulted to NaN. The built-in method ffill() and bfill() are commonly used to perform forward filling or backward filling to replace NaN.
- To resample a year by quarter and forward filling the values. The forward fill method ffill() will use the last known value to replace NaN.

value	
2012	1
2013	2
2014	3

value	
2012	1
2013	2
2014	3

Resample ('Q')

value	
2012Q1	1.0
2012Q2	NaN
2012Q3	NaN
2012Q4	NaN
2013Q1	2.0
2013Q2	NaN
2013Q3	NaN
2013Q4	NaN
2014Q1	3.0
2014Q2	NaN
2014Q3	NaN
2014Q4	NaN

ffill()

value	
2012Q1	1
2012Q2	1
2012Q3	1
2012Q4	1
2013Q1	2
2013Q2	2
2013Q3	2
2013Q4	2
2014Q1	3
2014Q2	3
2014Q3	3
2014Q4	3

III

Parameters Tuning for Time Series Analysis

Data Science Academy

HYPERPARAMETER TUNING FB PROPHET

- Cross-validation can be used for tuning hyperparameters of the model, such as changepoint_prior_scale and seasonality_prior_scale. A Python example is given below, with a 4x4 grid of those two parameters, with parallelization over cutoffs. Here parameters are evaluated on RMSE averaged over a 30-day horizon, but different performance metrics may be appropriate for different problems

```
import itertools
import numpy as np
import pandas as pd

param_grid = {
    'changepoint_prior_scale': [0.001, 0.01, 0.1, 0.5],
    'seasonality_prior_scale': [0.01, 0.1, 1.0, 10.0],
}

# Generate all combinations of parameters
all_params = [dict(zip(param_grid.keys(), v)) for v in itertools.product(*param_grid.values())]
rmses = []  # Store the RMSEs for each params here

# Use cross validation to evaluate all parameters
for params in all_params:
    m = Prophet(**params).fit(df)  # Fit model with given params
    df_cv = cross_validation(m, cutoffs=cutoffs, horizon='30 days', parallel="processes")
    df_p = performance_metrics(df_cv, rolling_window=1)
    rmses.append(df_p['rmse'].values[0])

# Find the best parameters
tuning_results = pd.DataFrame(all_params)
tuning_results['rmse'] = rmses
print(tuning_results)
```

HYPERPARAMETER TUNING FB PROPHET

The Prophet model has a number of input parameters that one might consider tuning. Here are some general recommendations for hyperparameter tuning that may be a good starting place.

- changepoint_prior_scale: This is probably the most impactful parameter. It determines the flexibility of the trend, and in particular how much the trend changes at the trend changepoints. If it is too small, the trend will be underfit and variance that should have been modeled with trend changes will instead end up being handled with the noise term. If it is too large, the trend will overfit and in the most extreme case you can end up with the trend capturing yearly seasonality. The default of 0.05 works for many time series, but this could be tuned; a range of [0.001, 0.5] would likely be about right. Parameters like this (regularization penalties; this is effectively a lasso penalty) are often tuned on a log scale.
- seasonality_prior_scale: This parameter controls the flexibility of the seasonality. Similarly, a large value allows the seasonality to fit large fluctuations, a small value shrinks the magnitude of the seasonality. The default is 10., which applies basically no regularization. That is because we very rarely see overfitting here (there's inherent regularization with the fact that it is being modeled with a truncated Fourier series, so it's essentially low-pass filtered). A reasonable range for tuning it would probably be [0.01, 10]; when set to 0.01 you should find that the magnitude of seasonality is forced to be very small. This likely also makes sense on a log scale, since it is effectively an L2 penalty like in ridge regression.
- holidays_prior_scale: This controls flexibility to fit holiday effects. Similar to seasonality_prior_scale, it defaults to 10.0 which applies basically no regularization, since we usually have multiple observations of holidays and can do a good job of estimating their effects. This could also be tuned on a range of [0.01, 10] as with seasonality_prior_scale.
- seasonality_mode: Options are ['additive', 'multiplicative']. Default is 'additive', but many business time series will have multiplicative seasonality. This is best identified just from looking at the time series and seeing if the magnitude of seasonal fluctuations grows with the magnitude of the time series (see the documentation here on multiplicative seasonality), but when that isn't possible, it could be tuned.

HYPERPARAMETER TUNING FOR NEURALPROPHET

- NeuralProphet has a number of hyperparameters that need to be specified by the user. If not specified, default values for these hyperparameters will be used.

- n_forecasts is the size of the forecast horizon. The default value of 1 means that the model forecasts one step into the future.

MODEL TRAINING RELATED PARAMETERS

- NeuralProphet is fit with stochastic gradient descent. If the parameter learning_rate is not specified, a learning rate range test is conducted to determine the optimal learning rate. The epochs and the loss_func are two other parameters that directly affect the model training process. If not defined, both are automatically set based on the dataset size.

- If it looks like the model is overfitting to the training data (the live loss plot can be useful hereby), you can reduce epochs and learning_rate, and potentially increase the batch_size. If it is underfitting, the number of epochs and learning_rate can be increased and the batch_size potentially decreased.

- The default loss function is the 'Huber' loss, which is considered to be robust to outliers. However, you are free to choose the standard MSE or any other PyTorch torch.nn.modules.loss loss function.

DATA PREPROCESSING RELATED PARAMETERS

- normalize_y is about scaling the time series before modelling. By default, NeuralProphet performs a (soft) min-max normalization of the time series. Normalization can help the model training process if the series values fluctuate heavily. However, if the series does not such scaling, users can turn this off or select another normalization.

- impute_missing is about imputing the missing values in a given series. Similar to Prophet, NeuralProphet too can work with missing values when it is in the regression mode without the AR-Net. However, when the autocorrelation needs to be captured, it is necessary for the missing values to be imputed, since then the modelling becomes an ordered problem. Letting this parameter at its default can get the job done perfectly in most cases.

.

Additional Resources

Data Science Academy

AR & MA MODELS

- Autoregressive AR process:
- Series current values depend on its own previous values
- AR(p) - Current values depend on its own p-previous values
- P is the order of AR process
- Moving average MA process:
- The current deviation from mean depends on previous deviations
- MA(q) - The current deviation from mean depends on q- previous deviations
- q is the order of MA process
- Autoregressive Moving average ARMA process

AUTOREGRESSION (AR)

- The autoregression (AR) method models the next step in the sequence as a linear function of the observations at prior time steps.
- The notation for the model involves specifying the order of the model p as a parameter to the AR function, e.g. AR(p). For example, AR(1) is a first-order autoregression model.
- The method is suitable for univariate time series without trend and seasonal components.

```
# AR example
from statsmodels.tsa.ar_model import AutoReg
from random import random
# contrived dataset
data = [x + random() for x in range(1, 100)]
# fit model
model = AutoReg(data, lags=1)
model_fit = model.fit()
# make prediction
yhat = model_fit.predict(len(data), len(data))
print(yhat)
```

MOVING AVERAGE (MA)

- The moving average (MA) method models the next step in the sequence as a linear function of the residual errors from a mean process at prior time steps.
- A moving average model is different from calculating the moving average of the time series.
- The notation for the model involves specifying the order of the model q as a parameter to the MA function, e.g. MA(q). For example, MA(1) is a first-order moving average model.
- The method is suitable for univariate time series without trend and seasonal components.

```
#MA example
from statsmodels.tsa.arima.model import ARIMA
from random import random
# contrived dataset
data = [x + random() for x in range(1, 100)]
# fit model
model = ARIMA(data, order=(0, 0, 1))
model_fit = model.fit()
# make prediction
yhat = model_fit.predict(len(data), len(data))
print(yhat)
```

AUTOREGRESSIVE MOVING AVERAGE (ARMA)

- The Autoregressive Moving Average (ARMA) method models the next step in the sequence as a linear function of the observations and residual errors at prior time steps.
- It combines both Autoregression (AR) and Moving Average (MA) models.
- The notation for the model involves specifying the order for the AR(p) and MA(q) models as parameters to an ARMA function, e.g. ARMA(p, q). An ARIMA model can be used to develop AR or MA models.
- The method is suitable for univariate time series without trend and seasonal components.

```
# ARMA example
from statsmodels.tsa.arima.model import ARIMA
from random import random
# contrived dataset
data = [random() for x in range(1, 100)]
# fit model
model = ARIMA(data, order=(2, 0, 1))
model_fit = model.fit()
# make prediction
yhat = model_fit.predict(len(data), len(data))
print(yhat)
```

AUTOREGRESSIVE INTEGRATED MOVING AVERAGE (ARIMA)

- The Autoregressive Integrated Moving Average (ARIMA) method models the next step in the sequence as a linear function of the differenced observations and residual errors at prior time steps.
- It combines both Autoregression (AR) and Moving Average (MA) models as well as a differencing pre-processing step of the sequence to make the sequence stationary, called integration (I).
- The notation for the model involves specifying the order for the AR(p), I(d), and MA(q) models as parameters to an ARIMA function, e.g. ARIMA(p, d, q). An ARIMA model can also be used to develop AR, MA, and ARMA models.
- The method is suitable for univariate time series with trend and without seasonal components.

```
# ARIMA example
from statsmodels.tsa.arima.model import ARIMA
from random import random
# contrived dataset
data = [x + random() for x in range(1, 100)]
# fit model
model = ARIMA(data, order=(1, 1, 1))
model_fit = model.fit()
# make prediction
yhat = model_fit.predict(len(data), len(data),
typ='levels')
print(yhat)
```

SEASONAL AUTOREGRESSIVE INTEGRATED MOVING-AVERAGE (SARIMA)

- The Seasonal Autoregressive Integrated Moving Average (SARIMA) method models the next step in the sequence as a linear function of the differenced observations, errors, differenced seasonal observations, and seasonal errors at prior time steps.
- It combines the ARIMA model with the ability to perform the same autoregression, differencing, and moving average modeling at the seasonal level.
- The notation for the model involves specifying the order for the AR(p), I(d), and MA(q) models as parameters to an ARIMA function and AR(P), I(D), MA(Q) and m parameters at the seasonal level, e.g. SARIMA(p, d, q)(P, D, Q)m where "m" is the number of time steps in each season (the seasonal period). A SARIMA model can be used to develop AR, MA, ARMA and ARIMA models.
- The method is suitable for univariate time series with trend and/or seasonal components.

```
# SARIMA example
from statsmodels.tsa.statespace.sarimax import SARIMAX
from random import random
# contrived dataset
data = [x + random() for x in range(1, 100)]
# fit model
model = SARIMAX(data, order=(1, 1, 1), seasonal_order=(0, 0, 0, 0))
model_fit = model.fit(disp=False)
# make prediction
yhat = model_fit.predict(len(data), len(data))
print(yhat)
```

BOX-JENKINS METHODS

- As we have seen ARIMA models have numerous parameters and hyper parameters, Box and Jenkins suggests an iterative three-stage approach to estimate an ARIMA model.
- Procedures:
    1. Model identification: Checking stationarity and seasonality, performing differencing if necessary, choosing model specification ARIMA(p, d, q).
    2. Parameter estimation: Computing coefficients that best fit the selected ARIMA model using maximum likelihood estimation or non-linear least-squares estimation.
    3. Model checking: Testing whether the obtained model conforms
    4. to the specifications of a stationary univariate process (i.e. the residuals should be independent of each other and have constant mean and variance). If failed go back to step 1

Thank you!

Data Science
Academy