-> Author: Rosario Scalia
-> Matriculation Number: 1000008648

Table of Contents

Introduction

Interest Domain

Time Series are an important class of data to analyze nowadays.

This kind of data can give useful insights in a wide spectrum of fields in order to understand what is the current and future behaviour of the studied system.

An important field related to Time Series are Finance.

In this scenario, it is crucial to understand and guess the trend of for example a Stock Market.

Knowing this insights provide several benefits like:

1. Choose the right investment with more security
2. Sell Stocks of a Company in a waning phase before company go in waning phase
3. Predict economical side-effects due to Stock Market Movements
4. Predict future value of a currency

An important Financial Index nowedays is the *Standard & Poor 500 (S&P)* index.

S&P is a is a free-float, capitalization-weighted index of the top 500 publicly listed stocks in the US (top 500 by market cap).

Having insights on this index can provide useful information about the trend of American Economy and ,for political reason, of the world economy.

Project Goal

The aim of this project is to analyze a Time Series dataset about some financial information (S&P index included) along the history.

The goals of analysis are extract meaningful insights from data and forecasting future S&P 500 Indexes.

Another sub-goal of this project is to understand base concepts for analyzing a Time Series dataset.

Dataset

Data Source and Description

Analyzed data was released in 2018 by *Data Hub* [7].

Topic of dataset is Finance, in particular data lake contains several measurements about top 500 leading publicly traded companies in the U.S.

Dataset is a Multi-Variate time series one having 1,769 data points.

Data was collected on monthly basis by *Robert Shiller* and ranges from 1871 to 2018.

Below, we report a brief note of the author about the procedure of data collection:

Stock market data used in my book, Irrational Exuberance [Princeton University Press 2000, Broadway Books 2001, 2nd ed., 2005] are available for download, Excel file (xls). This data set consists of monthly stock price, dividends, and earnings data and the consumer price index (to allow conversion to real values), all starting January 1871.

The price, dividend, and earnings series are from the same sources as described in Chapter 26 of my earlier book (Market Volatility [Cambridge, MA: MIT Press, 1989]), although now I use monthly data, rather than annual data. Monthly dividend and earnings data are computed from the S&P four-quarter totals for the quarter since 1926, with linear interpolation to monthly figures. Dividend and earnings data before 1926 are from Cowles and associates (Common Stock Indexes, 2nd ed. [Bloomington, Ind.: Principia Press, 1939]), interpolated from annual data.

Stock price data are monthly averages of daily closing prices through January 2000, the last month available as this book goes to press. The CPI-U (Consumer Price Index-All Urban Consumers) published by the U.S. Bureau of Labor Statistics begins in 1913; for years before 1913 1 spliced to the CPI Warren and Pearson's price index, by multiplying it by the ratio of the indexes in January 1913. December 1999 and January 2000 values for the CPI-U are extrapolated. See George F. Warren and Frank A. Pearson, Gold and Prices (New York: John Wiley and Sons, 1935). Data are from their Table 1, pp. 11–14.

For the Plots, I have multiplied the inflation-corrected series by a constant so that their value in january 2000 >equals their nominal value, i.e., so that all prices are effectively in January 2000 dollars.

Robert Shiller

Features of dataset is listed below:

Date	SP500	Dividend	Earnings	Consumer Price Index	Long Interest Rate	Real Price	Real Dividend	Real Earnings	PE10
Date	Number	Number	Number	Number	Number	Number	Number	Number	Number

- **SP500**, The S&P 500 Index, or Standard & Poor's 500 Index, is a market-capitalization-weighted index of 500 leading publicly traded companies in the U.S.

  It is not an exact list of the top 500 U.S. companies by market cap because there are other criteria to be included in the index.

  The index is regarded as one of the best gauges of large-cap U.S. equities.

  Another common U.S. stock market benchmark is the Dow Jones Industrial Average (DJIA).

  The S&P is a float-weighted index, meaning the market capitalizations of the companies in the index are adjusted by the number of shares available for public trading.

  Because it is widely considered the best gauge of large-cap U.S. equities, many funds are designed to track the performance of the S&P.

- **Dividend**, A dividend is the distribution of some of a company's earnings to a class of its shareholders, as determined by the company's board of directors.

  Common shareholders of dividend-paying companies are typically eligible as long as they own the stock before the ex-dividend date.

  Dividends may be paid out as cash or in the form of additional stock.

  Dividends are payments made by publicly listed companies as a reward to investors for putting their money into the venture.

- **Earnings**, A company's earnings are its after-tax net income.

  This is the company's bottom line or its profits.

  Earnings are perhaps the single most important and most closely studied number in a company's financial statements.

  It shows a company's real profitability compared to the analyst estimates, its own historical performance, and the earnings of its competitors and industry peers.

  Earnings are the main determinant of a public company's share price because they can be used in only two ways: They can be invested in the business to increase its earnings in the future, or they can be used to reward stockholders with dividends.

- **Consumer Price Index (CPI)**, The Consumer Price Index (CPI) is a measure that examines the weighted average of prices of a basket of consumer goods and services, such as transportation, food, and medical care.

  It is calculated by taking price changes for each item in the predetermined basket of goods and averaging them.

  Changes in the CPI are used to assess price changes associated with the cost of living.

  The CPI is one of the most frequently used statistics for identifying periods of inflation or deflation.

- **Long Interest Rate**, The interest rate is the amount a lender charges a borrower and is a percentage of the principal—the amount loaned.

  In current scenario, it was employed interest rate of 10 year maturity (period after debtor has to return capital to creditor) about U.S. Government Bonds.

- **Real Price**, real price of a company stocks is *Nominal Price* filtered respect inflation and other market side-effects.
- **Real Dividend**, real dividend of a company is *Nominal Dividend* filtered respect inflation and other market side-effects.
- **Real Earnings**, real earnings of a company is *Nominal Dividend* filtered respect inflation and other market side-effects.
- **PE10**, The P/E 10 (or CAPE) ratio is a valuation measure that uses real earnings per share (EPS) over a 10-year period to smooth out fluctuations in corporate profits that occur over different periods of a business cycle.

In this work, we focus on analyzing Standard & Poor historical data.

Exploratory Data Analysis

Descriptive Analysis

In [1]:
```python
#Import needed libraries
from os.path                                import join
from os                                     import listdir
from typing                                 import List
from pprint                                 import pprint
from functools                              import reduce
import pickle

import pandas                               as pd
from pandas                                 import Series,DataFrame
from pandas.plotting                        import lag_plot


import numpy                                as np
from numpy                                  import ndarray


import matplotlib.pyplot                    as plt


from scipy.stats                            import powerlaw, expon , gamma, t, chisquare, norm
from scipy.fftpack                          import fft,fftfreq


from statsmodels.graphics.gofplots          import qqplot
from statsmodels.graphics.tsaplots          import plot_acf
from statsmodels.tsa.stattools              import adfuller


from kats.consts                            import TimeSeriesData, SearchMethodEnum, TimeSeriesData
from kats.utils.decomposition               import TimeSeriesDecomposition
from kats.detectors.outlier                 import OutlierDetector
from kats.detectors.robust_stat_detection   import RobustStatDetector
from kats.detectors.seasonality             import ACFDetector,FFTDetector
import kats.utils.time_series_parameter_tuning as tpt

from kats.models.holtwinters                import HoltWintersModel , HoltWintersParams
from kats.models.sarima                     import SARIMAModel,SARIMAParams
from kats.models.prophet                    import ProphetModel, ProphetParams
from kats.utils.backtesters                 import BackTesterSimple, BackTesterRollingWindow , BackTesterExpandingWindow


from sklearn.cluster                        import KMeans
from sklearn.svm                            import SVC
from sklearn.naive_bayes                    import GaussianNB
from sklearn.linear_model                   import LogisticRegression
from sklearn.tree                           import DecisionTreeClassifier
from sklearn.ensemble                       import RandomForestClassifier
from sklearn.model_selection                import train_test_split
from sklearn.metrics                        import precision_score, recall_score , accuracy_score, confusion_matrix , ConfusionMatrixDisplay
from sklearn.mixture                        import GaussianMixture as GMM


import warnings
warnings.simplefilter(action='ignore')
```

In [2]:
```
#Program Costants
DATA_FILE_NAME:str      = "data_csv.csv"
DATA_PATH:str           = "data"
DATASET_PATH:str        = join(DATA_PATH , "s-and-p-500_zip" , "data" , DATA_FILE_NAME)
SEED:int                = 123
```

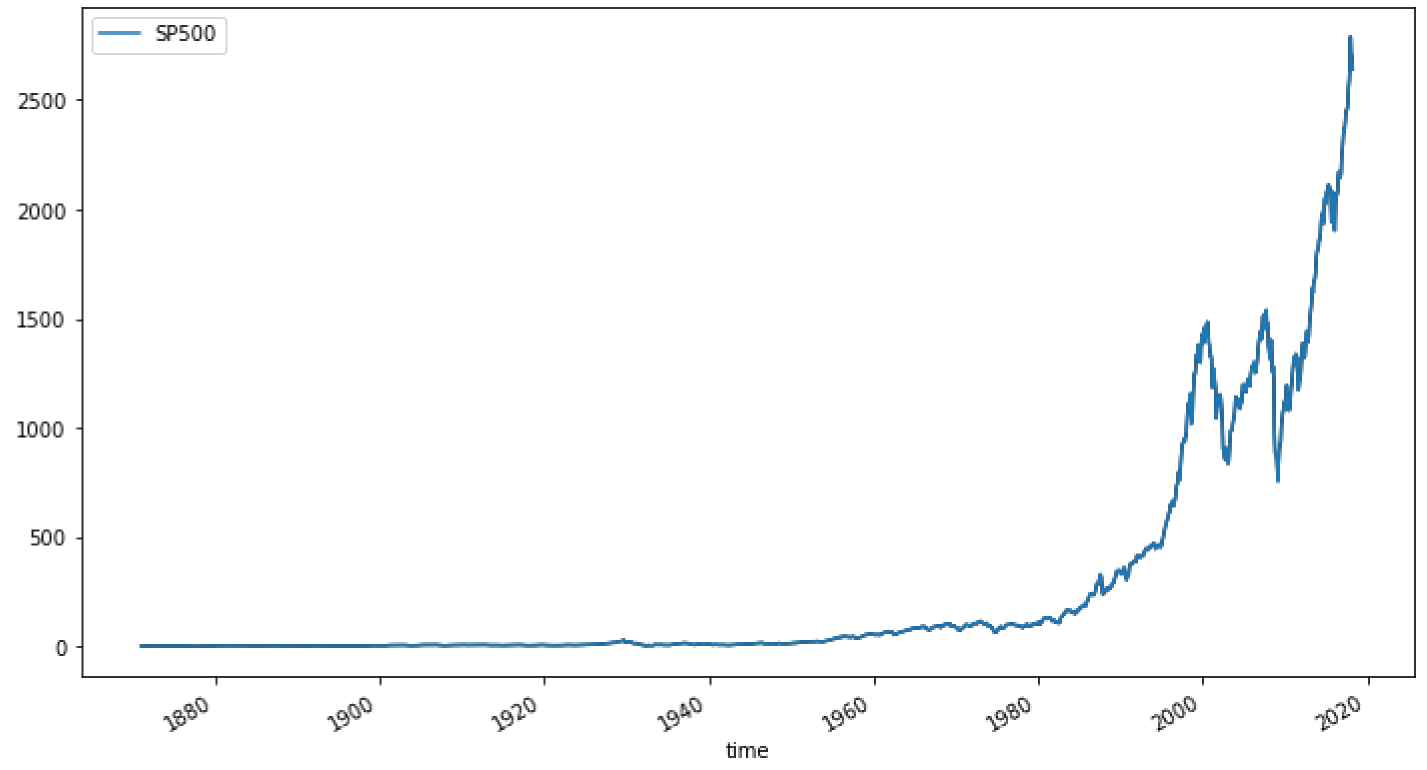
In [3]:
```
#Setup Seed
np.random.seed(123)
```

In [4]:
```
#Load Dataset from disk
try:

    #Wrapping SP500 Data in DataFrame Object
    full_dataset:DataFrame          = pd.read_csv(DATASET_PATH)
    sp_df:DataFrame                 = full_dataset[ ["Date","SP500"] ]
    sp_df:DataFrame                 = sp_df.rename({"Date":"time"} ,axis='columns')


    #Wrapping SP500 Data in Series Object
    sp:Series                       = Series(data=sp_df['SP500'].to_numpy() , index= pd.to_datetime( sp_df['time'] ) )


    #Wrapping SP500 Data in TimeSeries Object
    sp_series:TimeSeriesData        = TimeSeriesData(time=sp_df.time, value=sp_df.SP500)


    #Wrapping SP500 Data in Numpy Object
    sp_numpy:ndarray                = sp.to_numpy()
    sp_numpy:ndarray                = sp_numpy.astype('float64')
except Exception as exp:
    print("[ERROR] Unable to load dataset \n-> Reason: {}".format(exp))
```

In [5]:
```
#Basic Statistical Description of data
sp.describe()
```

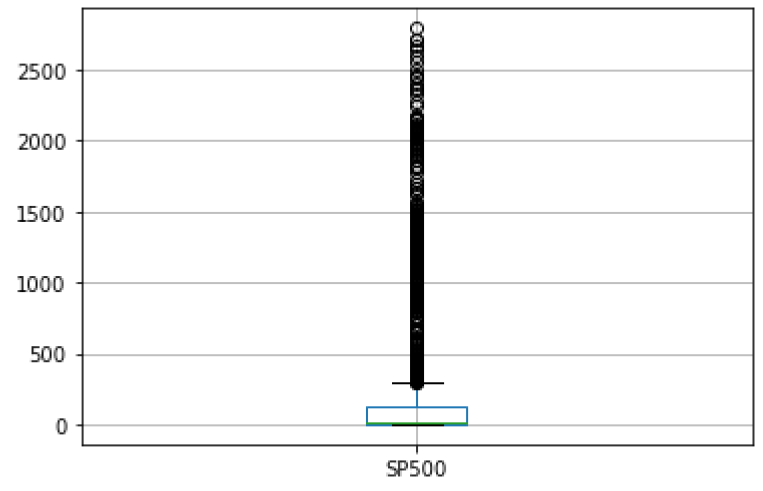
Out[5]:
```
count    1768.000000
mean      258.374570
std       514.103382
min         2.730000
25%         7.737500
50%        16.335000
75%       122.525000
max      2789.800000
dtype: float64
```

In [6]:
```
#Plot Time Series Data
sp_series.plot(cols=['SP500'])
plt.show()
```



From plot, we can see that series has a long period of stability between 1880-1975; after this period series has a trend phenomenon.

In [7]:
```
#Box-Plot SP500
sp_df[['SP500']].boxplot()
plt.show()
```



In [8]:
```
#Box-Plot Analysis without possible outliers
sp_df_data                      = sp_df['SP500']
q1,_,q3                         = sp_df_data.quantile([1/4,2/4,3/4])
outlier_thr:float               = q3+1.5*(q3-q1)

n_possible_outlier:int          = len( sp_df_data[ sp_df_data > outlier_thr  ] )
print("[!] Number of Possible Outliers: {}".format(n_possible_outlier))
total_size:int                  = len(sp_df)
perc_possible_outlier:int       = int(n_possible_outlier/total_size*100)

print("[!] Percentage of Possible Outlier for Standard & Poort Index: {} %".format(perc_possible_outlier))
```

```
[!] Number of Possible Outliers: 353
[!] Percentage of Possible Outlier for Standard & Poort Index: 19 %
```

Showed Box-plot underlines a compact population where first quantile is near to median.

Furthermore, there is 19 % of sample out-of-scale; these are probably outliers.

Nevertheless, we are dealing with temporale data; for this reason we need a more sophisticated procedure for detecting outliers.

In [9]:
```
#Skewness
sp_df.skew()
```

Out[9]:
```
SP500    2.39734
dtype: float64
```

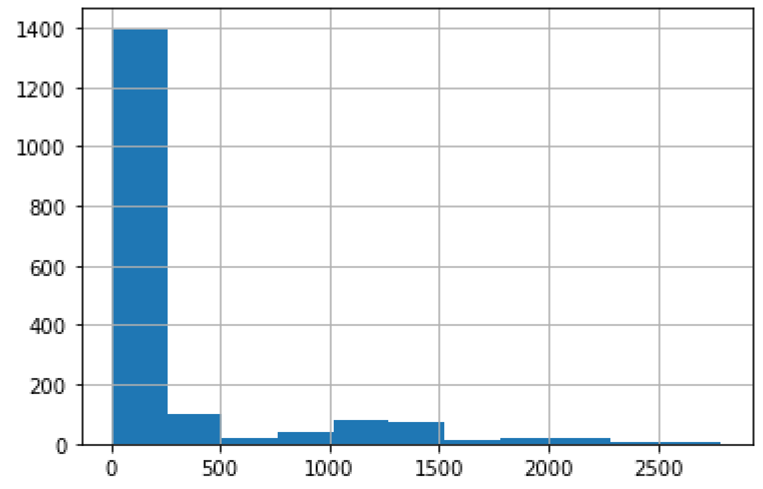
Skew for Standard & Poor index is greater than zero, this means that distribution is asymetric to left.

In [10]:
```
#Kurtosi
sp_df.kurtosis()
```

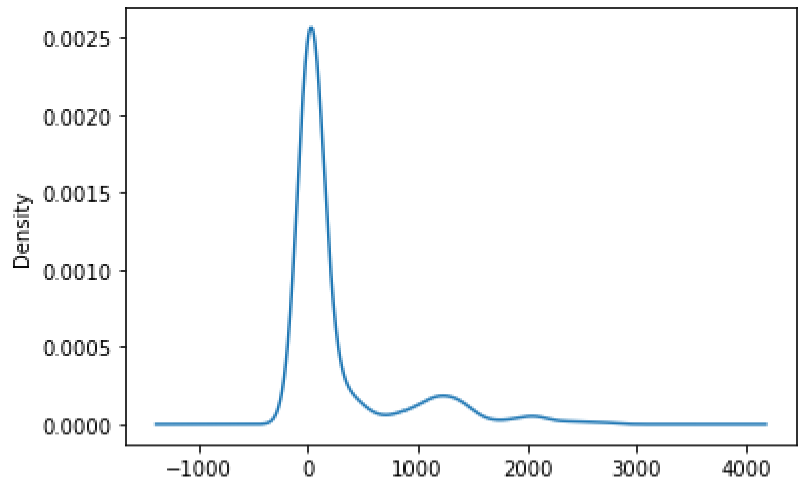
Out[10]:
```
SP500    5.214177
dtype: float64
```

Kurtosi indexes tells us that SP500 series is leptocurtica, more sharp than a Gaussian.

In [11]:
```
#Histogram
sp.hist(bins=11)
plt.show()
```



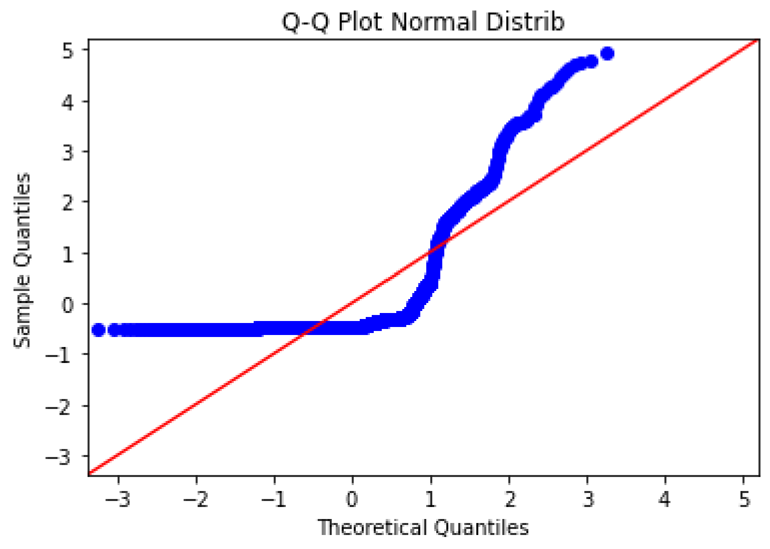
In [12]:
```
sp.plot(kind='kde')
plt.show()
```



```
In [13]: def chi_square_test(pDistrib:object, pSample:ndarray , pLabel:str , pNBin:int=11) -> None:
             """
             # **chi_square_test**

             Chi-Square Test Function

             Args:
                 pDistrib    (object)                    : Hypothized Distribution Object
                 pSample     (ndarray)                   : Sample
                 pLabel      (str)                       : Hypothized Distribution Name
                 pNBin       (int        | DEF = 11)     : number of test split. Defaults to 11.
             """

             # [1] Fit Data to Hyphotized Disttrib
             distrib_params:tuple        = pDistrib.fit(pSample)
             distrib_f:object            = pDistrib(*distrib_params)

             # [2] Compute Test Suddivisions
             num_par:int                 = len(distrib_params)
             edges:ndarray               = np.linspace(distrib_f.ppf(0.01),distrib_f.ppf(0.99),pNBin)

             # [3] Eval Theoretical Freqs
             cumulative_probs:ndarray    = distrib_f.cdf(edges)
             probs:ndarray               = np.diff(cumulative_probs)
             th_fr:ndarray               = probs*len(pSample)

             # [4] Eval Empiric Freq
             emp_fr, _                   = np.histogram(pSample,edges)

             # [5] Check for prerequisites of the Test
             if th_fr.min()>=5 or emp_fr.min()>=5:
                 print("[!] Chi-Square Test assuming {} Theoretical Distribution \n-> Unable to Perform Test for poor number of samples (a.k.a. freq < 5)".format( pLabel))
                 return

             # [6] Executes Test
             stat , pvalue               = chisquare(emp_fr,th_fr,ddof=num_par)

             print("[!] Chi-Square Test assuming {} Theoretical Distribution \n-> Statistic: {} \n-> p-value: {} \n-> N Bins: {}".format( pLabel ,stat,pvalue , pNBin))
```

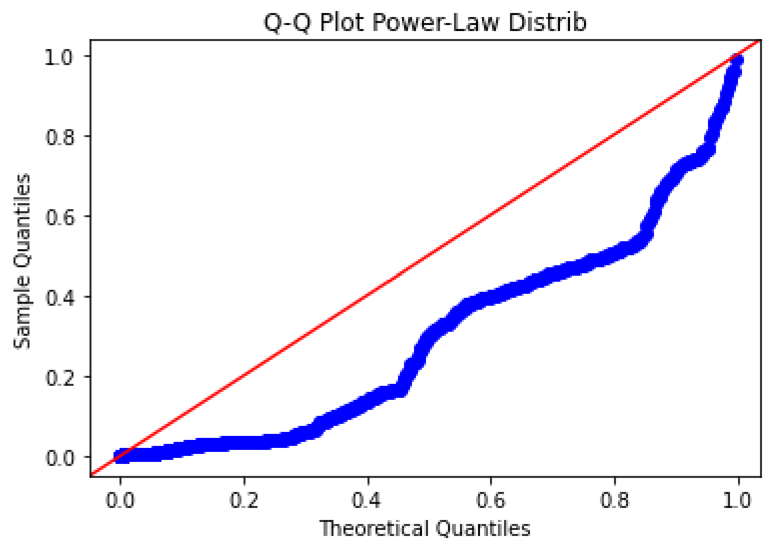
In [14]:
```
chi_square_test(norm , sp_numpy , "Normal" )

qqplot(sp_numpy, fit=True, line='45', dist=norm)
plt.title("Q-Q Plot Normal Distrib")
plt.show()
```

```
[!] Chi-Square Test assuming Normal Theoretical Distribution
-> Unable to Perform Test for poor number of samples (a.k.a. freq < 5)
```



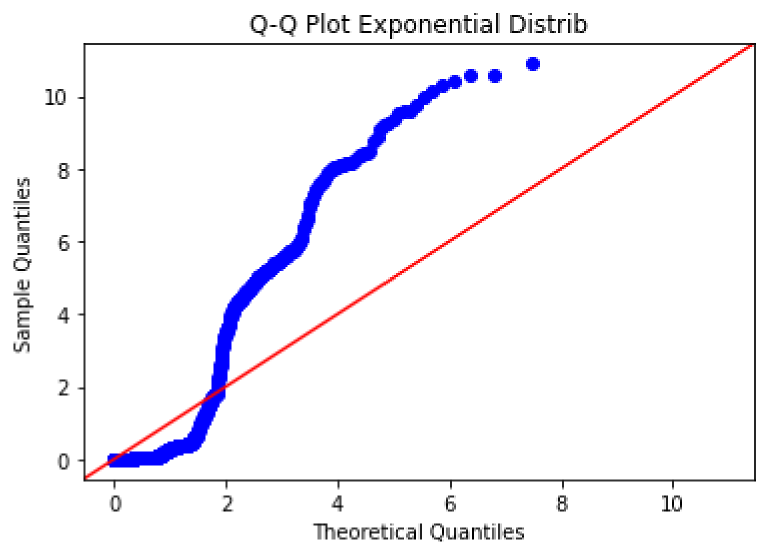
In [15]:
```
chi_square_test(powerlaw , sp_numpy , "Power-Law" )

qqplot(sp_numpy, fit=True, line='45', dist=powerlaw)
plt.title("Q-Q Plot Power-Law Distrib")
plt.show()
```

```
[!] Chi-Square Test assuming Power-Law Theoretical Distribution
-> Unable to Perform Test for poor number of samples (a.k.a. freq < 5)
```



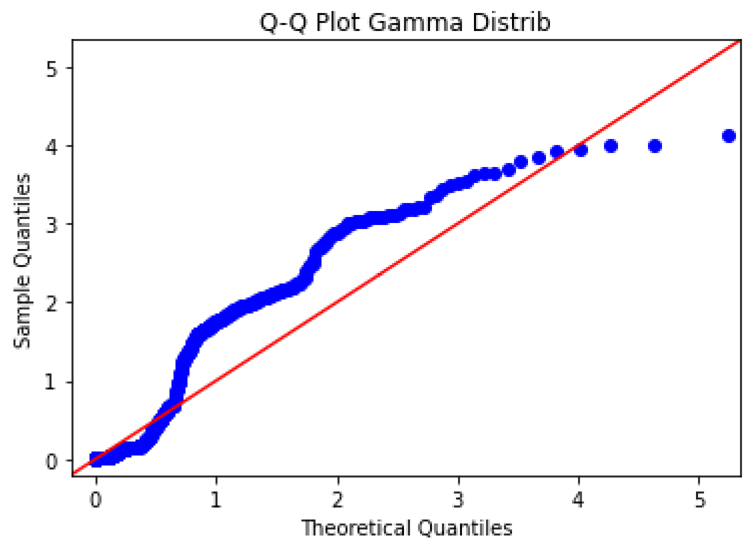
```
In [16]: chi_square_test(expon , sp_numpy , "Exponential" )
         qqplot(sp_numpy, fit=True, line='45', dist=expon)
         plt.title("Q-Q Plot Exponential Distrib")
         plt.show()
```

```
[!] Chi-Square Test assuming Exponential Theoretical Distribution
-> Unable to Perform Test for poor number of samples (a.k.a. freq < 5)
```



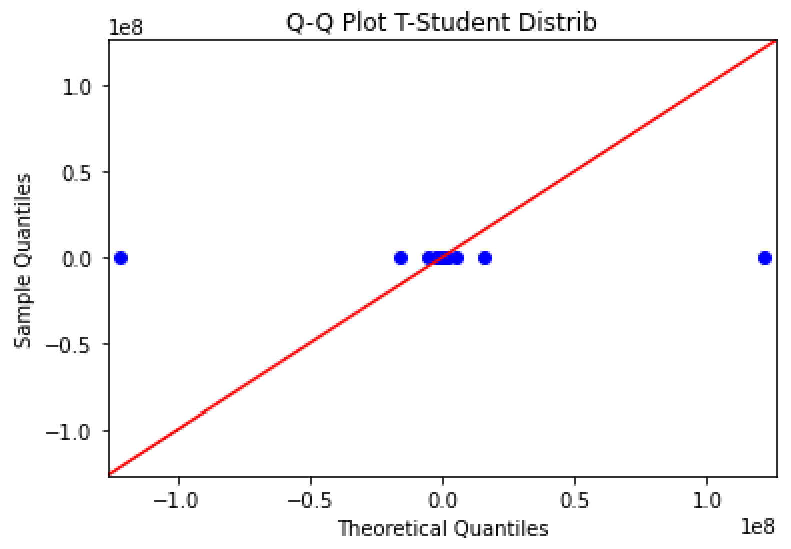
```
In [17]: chi_square_test(gamma , sp_numpy , "Gamma" )

         qqplot(sp_numpy, fit=True, line='45', dist=gamma)
         plt.title("Q-Q Plot Gamma Distrib")
         plt.show()
```

```
[!] Chi-Square Test assuming Gamma Theoretical Distribution
-> Unable to Perform Test for poor number of samples (a.k.a. freq < 5)
```



In [18]:
```
chi_square_test(t , sp_numpy , "T" )

qqplot(sp_numpy, fit=True, line='45', dist=t)
plt.title("Q-Q Plot T-Student Distrib")
plt.show()
```

```
[!] Chi-Square Test assuming T Theoretical Distribution
-> Statistic: 480.09331828813526
-> p-value: 1.6301975888329255e-100
-> N Bins: 11
```



Showed Chi-Square tests underlines a difficult to assign a well known distribution to data; this is caused by the temporal relation between data.

Time Series Decomposition

A useful abstraction for selecting forecasting methods is to break a time series down into systematic and unsystematic components.

- Systematic: Components of the time series that have consistency or recurrence and can be described and modeled.
- Non-Systematic: Components of the time series that cannot be directly modeled.

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.

A series is thought to be an aggregate or combination of these four components.

All series have a level and noise. The trend and seasonality components are optional.

It is helpful to think of the components as combining either additively or multiplicatively.

An additive model suggests that the components are added together as follows:

$y(t) = Level + Trend + Seasonality + Noise$

An additive model is linear where changes over time are consistently made by the same amount.

A linear trend is a straight line.

A linear seasonality has the same frequency (width of cycles) and amplitude (height of cycles).

A multiplicative model suggests that the components are multiplied together as follows:

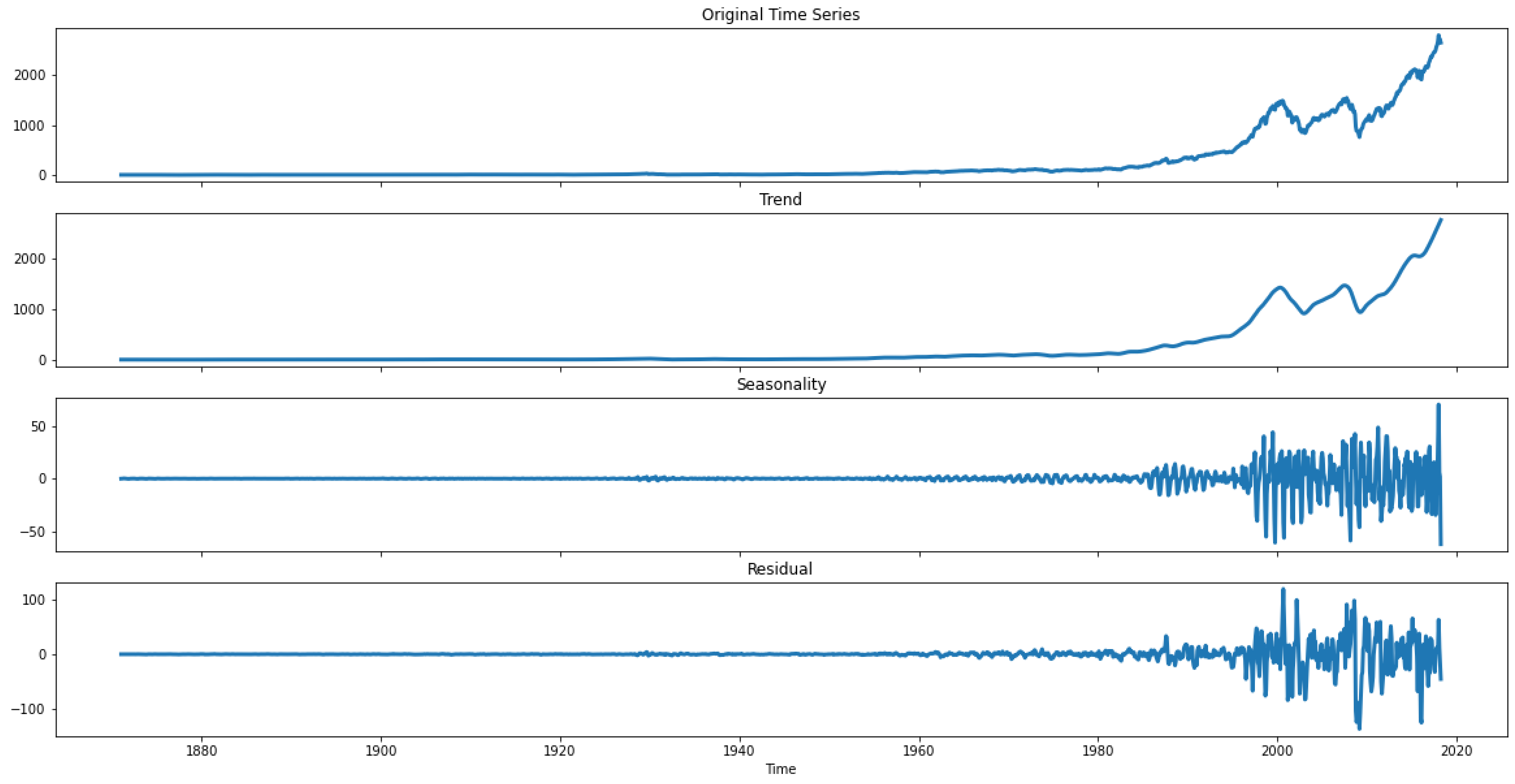
$y(t) = Level \times Trend \times Seasonality \times Noise$

A multiplicative model is nonlinear, such as quadratic or exponential. Changes increase or decrease over time.

A nonlinear trend is a curved line.

A non-linear seasonality has an increasing or decreasing frequency and/or amplitude over time.

In [19]:
```
decomposer:TimeSeriesDecomposition = TimeSeriesDecomposition(sp_series)
decomposition:dict  = decomposer.decomposer()
decomposer.plot()
plt.show()
```

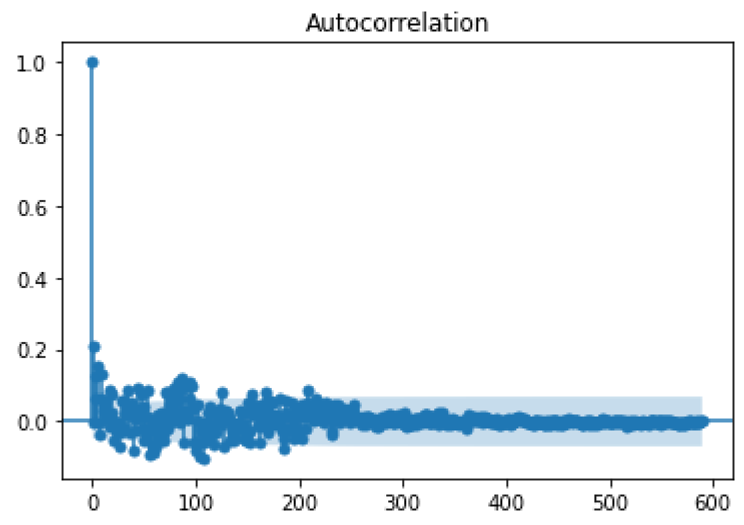


Showed decomposition underlines the following facts:

1. Series has a strong trend at his end period
2. Series has a low seasonal component if any
3. Series has a noise component

```
In [20]: detector        = ACFDetector(sp_series)
         acf_dt_outcome  = detector.detector(diff=1,alpha = 0.01)
         print(acf_dt_outcome)
         detector.plot()
```

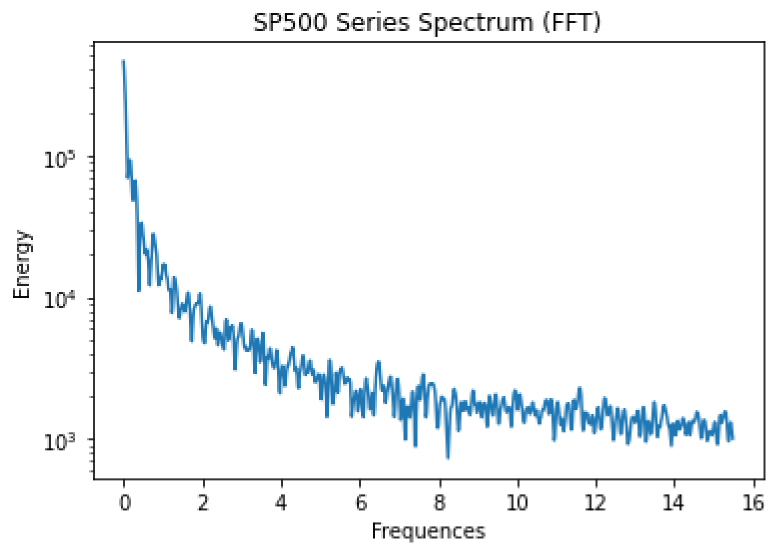
```
{'seasonality_presence': True, 'seasonalities': [3, 4, 5, 19, 34, 43, 53, 71, 82, 83, 91, 94]}
```



```
In [21]: fft_detect = FFTDetector(sp_series)
         fft_dt_outcome = fft_detect.detector(sample_spacing=31.0)
         print(fft_dt_outcome)
```

```
{'seasonality_presence': False, 'seasonalities': []}
```

In [22]:
```
# Plot FFT of Series using Scipy
fft_sp_numpy              = fft(sp_numpy)
energy_fft_sp_numpy       = abs(fft_sp_numpy)
freqs                     = fftfreq(sp_numpy.shape[0],1/31)

plt.semilogy(freqs[0:int(freqs.size/2)], energy_fft_sp_numpy[0:int(energy_fft_sp_numpy.size/2)])
plt.title("SP500 Series Spectrum (FFT)")
plt.xlabel("Frequences")
plt.ylabel("Energy")
plt.show()
```

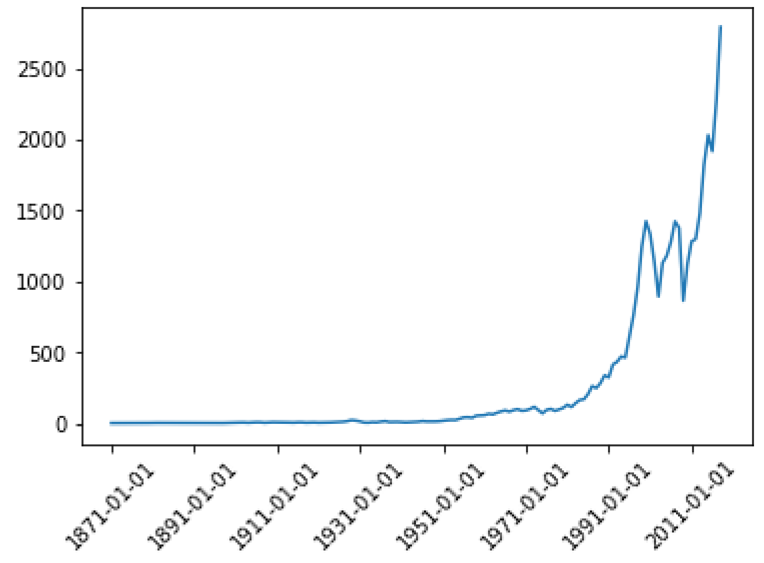


Detection of period for Seasonality underlines the following aspects:

1. ACF Detector underlines a set of Seasonality Periods
2. Fourier Detector (and his Graph) underlines no spikes in corrispondence of ACF Detector seasonality periods; for this reason we concludes that Series doesn't have a predominant Seasonal Component and that ACF Detector outcomes are statistically insignificant.

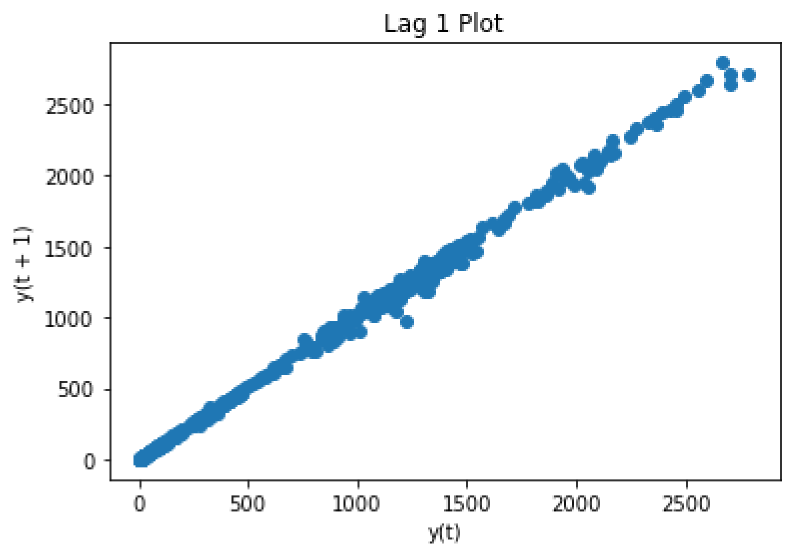
Data Visualization

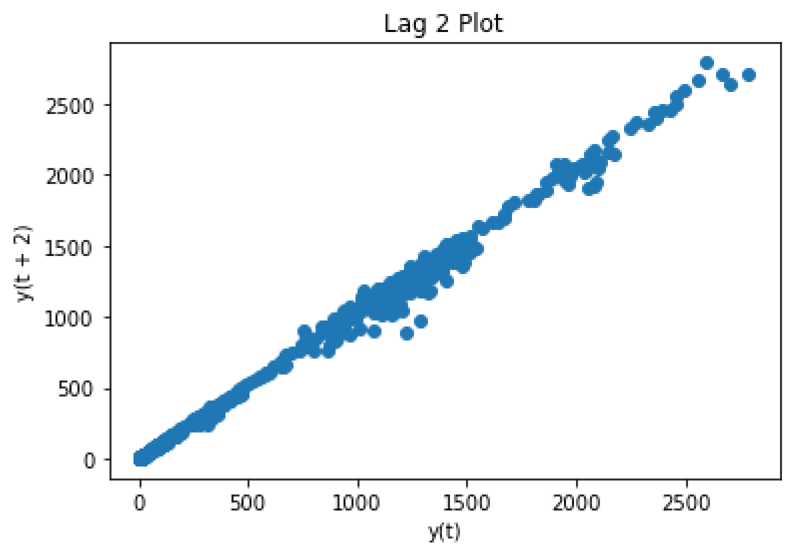
```
In [23]: sp_df_numpy:ndarray               = sp_df.to_numpy()
         sp_list:List[tuple]               = sp_df_numpy.tolist()
         annual_data:List[tuple]           = list( filter( lambda x: x[0].split("-")[1] == "01" , sp_list ) )

         time_index:List[str]              = list( map( lambda x:x[0] , annual_data ) )
         sp_annual_data:List[float]        = list( map( lambda x:x[1] , annual_data ) )

         annual_data_series:List[tuple]  = Series(data=sp_annual_data , index=time_index )

         annual_data_series.plot()
         plt.xticks(rotation=45)
         plt.show()
```

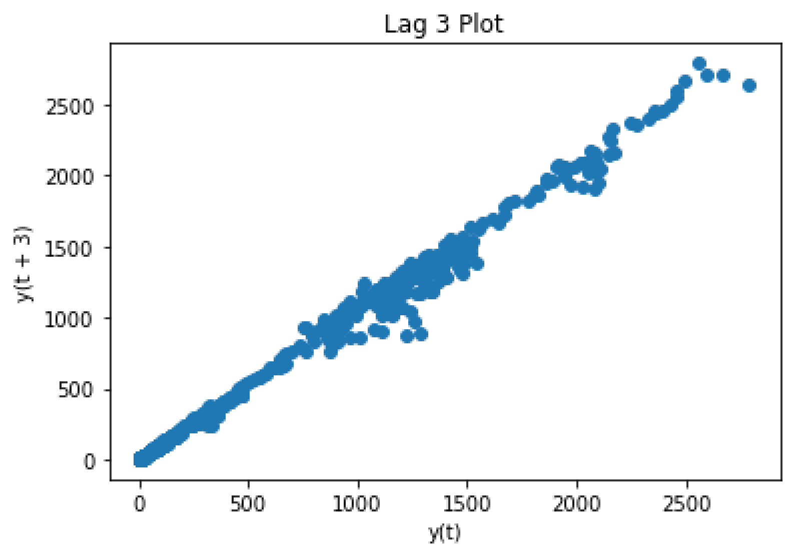


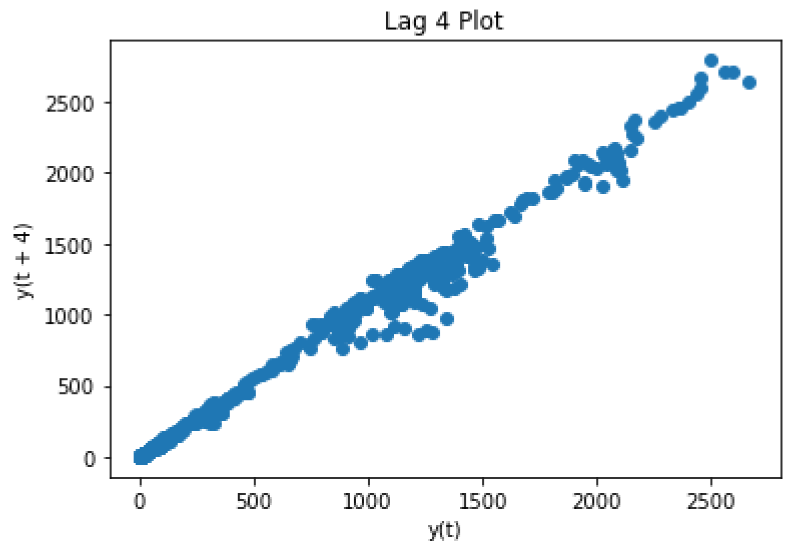
Annual Time Series data doesn't show a significant change respect original monthly time series data

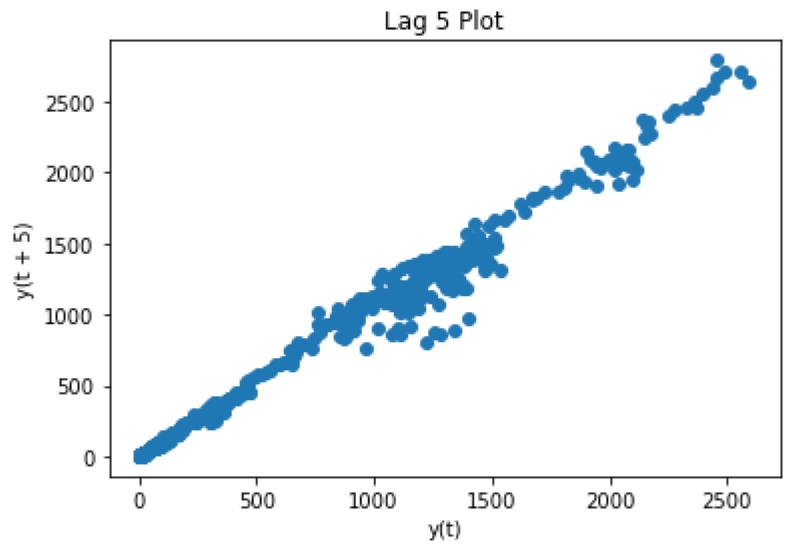
In [24]:
```
#Lag-Plot
lag_plot(sp,lag=1)
plt.title("Lag 1 Plot")
plt.show()

lag_plot(sp,lag=2)
plt.title("Lag 2 Plot")
plt.show()

lag_plot(sp,lag=3)
plt.title("Lag 3 Plot")
plt.show()

lag_plot(sp,lag=4)
plt.title("Lag 4 Plot")
plt.show()

lag_plot(sp,lag=5)
plt.title("Lag 5 Plot")
plt.show()
```











Lag plots show a positive correlation up to lag 5 for our Series.

Stationary Analysis

Stationarity of a time series is a very welcome property in order to fit a good Forecast Model.

Concern of Stationarity is quite related to Unit Roots of a Stochastic Process (mathematical model behind time series).

In probability theory and statistics, a unit root is a feature of some stochastic processes that can cause problems in statistical inference involving time series models.

A linear stochastic process has a unit root if 1 is a root of the process's characteristic equation.

Such a process is non-stationary but does not always have a trend.

If the other roots of the characteristic equation lie inside the unit circle—that is, have a modulus (absolute value) less than one—then the first difference of the process will be stationary; otherwise, the process will need to be differenced multiple times to become stationary.

If there are d unit roots, the process will have to be differenced d times in order to make it stationary.

Due to this characteristic, unit root processes are also called difference stationary.

In order to understand if a time series is stationary, there was developed several techniques the spans from Data Visualizaziont to Statistical Tests.

In our work, we uses two metodologies tailored for category:

1. ACF Plot (Data Visualization)
2. Augmented Dickey–Fuller (Statistic Test)

AutoCorrelation (ACF) measure serial (or self) correlation of a times series, in particular measure correlation between time series against herself lagged version.

Augmented Dickey–Fuller is a Unit Root test having the following charateristic:

1. Null Hypothesis (H0) is that process has Unit Roots
2. Alternative Hypothesis (H1) is that process doesn't have Unit Roots
3. Statistic Computation involves the following steps:

   1) Assumes that time series is modelled by an $AR^{(p)}(\cdot)$ model

   2) Evaluate difference $\Delta X_t = AR^{(p)}(X_t) - AR^{(p)}(X_{t-1})$

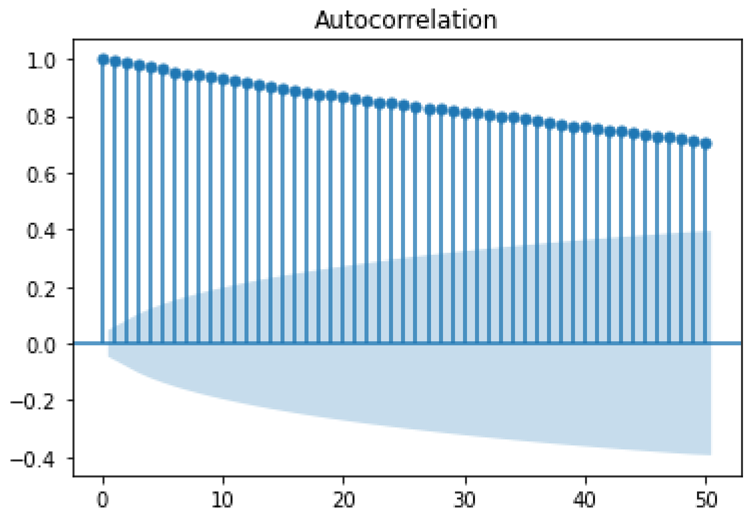
   3) Fit Leaste Square Model on ΔX_t

   4) Build Statistic: $\tau = \frac{\hat{\phi}_1}{SE(\hat{\phi}_1)}$ where SE is Standard Error $SE = (\sum_{t=2}^{n} \frac{(\Delta X_t - AR^{(p)}(X_t))^2}{n-3}) \cdot (\sum_{t=2}^{n} (X_{t-1} - \bar{X})^2)^{-1/2}$ and $\hat{\phi}_1$ is first estimated Model Paramiter.

   5) Dickey and Fuller derived the limit distribution as $n \to \infty$ of the τ ratio under the unit root assumption ($\hat{\phi}_1 = 0$)

   6) The 0.01, 0.05, and 0.10 quantiles of the limit distribution of τ are −3.43, −2.86, and −2.57, respectively.

```
In [25]: plot_acf(sp,lags=50)
         plt.show()
```

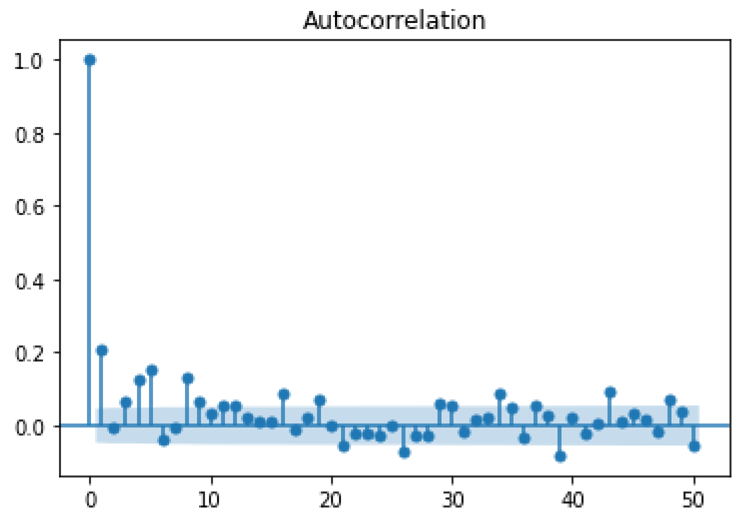


```
In [26]: result = adfuller(sp)
         print('ADF Statistic: %f' % result[0])
         print('p-value: %f' % result[1])
         print('Critical Values:')
         for key, value in result[4].items():
             print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: 3.603097
p-value: 1.000000
Critical Values:
        1%: -3.434
        5%: -2.863
        10%: -2.568
```

```
In [27]: diff_1 = sp.diff().dropna()
```

```
In [28]: plot_acf(diff_1,lags=50)
         plt.show()
```



```
In [29]: result = adfuller(diff_1)
         print('ADF Statistic: %f' % result[0])
         print('p-value: %f' % result[1])
         print('Critical Values:')
         for key, value in result[4].items():
             print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -7.901572
p-value: 0.000000
Critical Values:
	1%: -3.434
	5%: -2.863
	10%: -2.568
```

Original Time Series is not stationary because ACF "don't drops quickly" and ADF Statistic is greater than levels of significance.

Instead, One-Order Differenced Time Series is stationary because ACF drops quickly and ADF Statistic is lower than levels of significance.

Outlier Detection

Outlier detection algorithm works as follows:

1) Seasonal decomposition of the input time series, with additive or multiplicative decomposition as specified.

2) Generate a residual time series by either removing only trend or both trend and seasonality if the seasonality is strong.

3) Detect points in the residual which are outside 3 times the inter quartile range.

```
In [30]: #Outlier removing from Standard & Poor Index
         sp_outliers_detector = OutlierDetector(sp_series, 'additive')
         sp_outliers_detector.detector()
```

```
In [31]: n_outlier_sp500      = len(sp_outliers_detector.outliers[0])
         print("[!] Number of Outliers detected for Standard & Poor Index: {}".format(n_outlier_sp500))
```

```
[!] Number of Outliers detected for Standard & Poor Index: 415
```

```
In [32]: sp_cleaned          = sp_outliers_detector.remover(interpolate = True)
```

The procedure used for removing outliers is more robust than the first one employed in this notebook; infact last procedure employs a decomposition in order to remove noise and temporal-effects from the series.

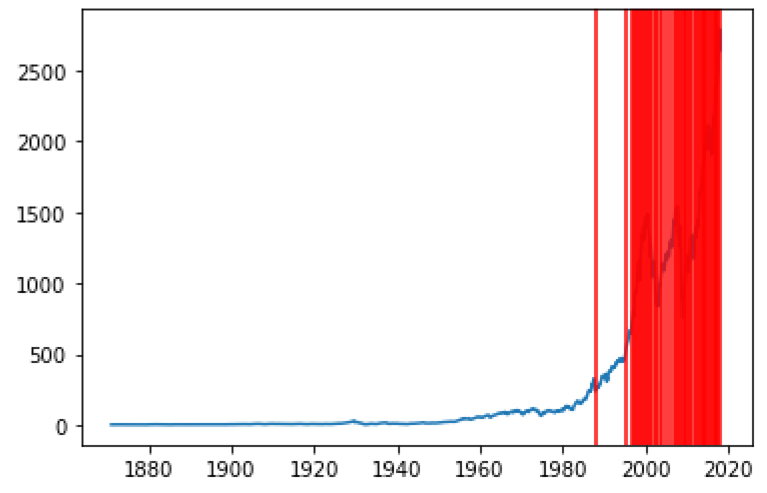
After that, analysis is similar to first approach of this notebook.

Changepoints Detection

RobustStatDetector is a change point detection algorithms that finds mean shifts. It works as follows:

1. Smooth the time series using a moving average
2. Calculate the differences in the smoothed time series over a fixed number of points.
3. Calculate the z-scores and p-values for the differences calculated in step 2.
4. Return points where the p-value is smaller than a prescribed threshold

In [33]:
```python
detector:RobustStatDetector = RobustStatDetector(sp_series)
changepoints:List[tuple]     = detector.detector(p_value_cutoff = 1e-1, comparison_window=2)

detector.plot(changepoints)
plt.show()
```



Algorithm returns a list of changepoints; surely first changepoint detected (near 1985) is real.

For the others, we need further investigation in order to check the significance level.

Forecast Experiments

Seasonal ARIMA

Seasonal ARIMA (Autoregressive-Integrated-Moving Average) is a statistical model for analyzing and forecasting time series data and more in generale Stochastic Processes.

The Model is a composition of two further ones:

1. Autoregressive Model
2. Moving Average Model

Autoregressive Model (AR) is a linear model that describes the outcome of a Stochastic Process as a linear combination of its own previous values and a stochastic term (an imperfectly predictable term); thus the model is in the form of a stochastic difference equation.

The analytic form of the AR model is the following:

$$F_t \;=\; \mathrm{AR}_p(t) \;=\; c + \varepsilon_t + \sum_{i=1}^{p} \phi_i \cdot X_{t-i}$$

The above Formula shows the following cues:

1. $\mathbf{F_t}$ is the predicted value of time serie at time t given the last p values of time serie
2. $\mathbf{c}$, costant value

3. $\phi_{\mathbf{i}}$, parameters of the model
4. $\mathbf{p}$, hyperparamiter that set the number of past observations in order to perform inference for the next step. For that, the model is usually referred as AR(p) where p is the aforementioned hyperparam
5. $\varepsilon_{\mathbf{t}}$, white noise term that model the error
6. $\mathbf{X_t}$ is the actual value of time time at step t

An autoregressive model can thus be viewed as the output of an *all-pole infinite impulse response filter* whose input is white noise.

Moving Average Model (MA) is the other sub-part of SARIMA Model.

The moving-average model specifies that the output variable depends linearly on the current and various past values of a stochastic (imperfectly predictable) term.

The analytic form of MA model is the following:

$$F_t \;=\; \mathrm{MA}_q(t) \;=\; \mu + \varepsilon_t + \sum_{i=1}^{q} \theta_i \cdot \varepsilon_{t-i}$$

The formula shows the following features:

1. $\mathbf{F_t}$ is the predicted value of time serie at time t given the last q random shocks suffered by the system
2. μ is the mean of the time serie
3. $\varepsilon_{\mathbf{t-i}}$ is the random shock (or error term) occured at time t-i
4. $\theta_{\mathbf{i}}$ are the parameters of the Model
5. $\mathbf{q}$ is the hyperparameter of the model, it rules the number of past random shocks to employ for the inference of the next step of the time serie. For that, MA model is usually referred as MA(q) model.
6. $\mathbf{X_t}$ is the actual value of time time at step t

Random shocks at each point are assumed to be mutually independent and to come from the same distribution, typically a normal distribution, with location at zero and constant scale.

From a Signal Processing perspective, the moving-average model is a finite impulse response filter applied to white noise.

Seasonal ARIMA is a combination of these two models enriched with a *specific pre-processing* of data in order to ensure the *weak stationarity* of the Stochastic Process induced by the time serie.

Before explaining the pre-processing needed, we briefly report the definition of Weak Stationarity.

A Stochastic Process X is Weak Stationary if and only if:

1. $\mu(X_t) = \mu(X_t + \tau) \quad \forall \tau \in \mathbb{R}$
2. $Cov_{X,X}\,(t_1, t_2) = Cov_{X,X}\,(t_1 - t_2, 0) \quad \forall t_1, t_2 \in \mathbb{R}$
3. $\mathbb{E}\,[|X_t|^2] < \infty \quad \forall t \in \mathbb{R}$

In essence, the definition is telling us that the mean of the time serie not vary respect the time.

Said that, it is worth saying that AR, MA models and ARMA (combination of AR and MA) are models designed for Weak Stationary Stochastic Process (a.k.a. Weak Stationary Time Series).

Usually, real time series are not stationary, for this reason it would take a way to transform.

The method is the *Differencing* and corresponde to the letter "I" of the SARIMA model.

Differencing involves the following operation:

$$z_t \;=\; \nabla X_t \;=\; X_t - X_{t-1}$$

This operation ,essentialy, removes *Trends* from the time serie and enforce Weak Stationarity of hers.

Furthermore, is worth notice that data can require more steps of differencing in order to reach the weak stationarity.

In all cases, an ARMA model applied to the differenced data is usually referred as *ARIMA* model.

From a Signalt Theory point of view, Differencing acts as high-pass filter to the original signal.

Another important issue of time series data is *Seasonality*.

Usually, some specific behaviour of the Stochastic Process happpens at cyclic intervals of time (think at the grow of purchases in corrispondance of the Christmas Holidays).

This issue is called *Seasonality* and must be put in the accout in order to perform good forecast.

Therefore, we properly combine two ARIMA models (one for regular data and one for Seasonal Data extracted from the studied time serie) in the Seasonal ARIMA model.

This model has the following analytic expression:

$$F_t \;=\; \mathrm{SARIMA}^{\,(p,d,q)\times(P,D,Q)_s} \;=\; \left[\mathrm{MA}_Q^{(seas)}(t)\cdot \mathrm{MA}_q^{(reg)}(t)\right] - \left[\mathrm{AR}_P^{(seas)}(t)\cdot \mathrm{AR}_p^{(reg)}(t)\right]$$

This model is usually referred as $\mathrm{SARIMA}^{\,(p,d,q)\times(P,D,Q)_s}$ and has the following hyperparams:

1. $\mathbf{p}$, number of past observation for the computation of the Regular AR Model
2. $\mathbf{d}$, number of differecing performed to the Regular Data
3. $\mathbf{q}$, number of past random shocks for the computation of the Regular MA Model
4. $\mathbf{P}$, number of past observation for the computation of the Seasonal AR Model
5. $\mathbf{D}$, number of differecing performed to the Seasonal Data extracted from the Regular Data
6. $\mathbf{Q}$, number of past random shocks for the computation of the Seasonal MA Model
7. $\mathbf{s}$, period of a season

At this point, one can ask how to compute optimal SARIMA hyper params value.

Well, there is 3 ways:

1. Plotting Autocorrelation (AFCF) and Partial Autocorrelation (PACF) of the data and try to guess the right model with the help of some *common sense* rules.
2. Optimizing a specific criterion respect to the hyperparamiters (ex. BIC, AIC etc..)
3. Perform GridSearch on a restricted space of hyperparamiters

Holt-Winters

Holt-Winters is a forecasting algorithm based on the following three techniques of smoothing:

1. Weighted Average
2. Exponential Smoothing
3. Holt Exponential Smoothing

Weighted Average is a simple mean where each observation is weighted with a specific weight; the denominator of the mean is the sum of all weights.

Usually, the weight of a Weighted Average are modelled throught specific mathematical functions.

Exponential Smoothing is a weighted average where weights decay exponentials from the most recent to the oldest historical value.

In this setup, you make the assumption that recent values in the time serie are most important than old ones in order to perform forecasting.

This assumption is deleterious in case of time series that show Trend and Seasonality.

Holt Exponential Smoothing is an arrangment of Exponential Smoothing in order to fix the Trend problem aforementioned.

Holt-Winters Algorithm is an improvement of Holt in order to fix the problems relative to Seasonality.

The algorithm is declined in two flavours respect to the "type" of Seasonality: *Additive* or *Multiplicative*.

In case of Multiplicative Seasonality, the algorithm induce the following Recursive set of Equations:

$$\begin{cases} F_{t+m} & = (s_t + m \cdot b_t) \cdot c_{t-L+m \ mod \ L} \\ s_t & = \alpha \cdot \frac{X_t}{c_{t-L}} + (1-\alpha) \cdot (s_{t-1} + b_{t-1}) \\ b_t & = \beta \cdot (s_t - s_{t-1}) + (1-\beta) \cdot b_{t-1} \\ c_t & = \gamma \cdot \frac{X_t}{s_t} + (1-\gamma) \cdot c_{t-L} \end{cases}$$

The above set of equations employ the following symbols:

1. $\mathbf{F_{t+m}}$, forecasting at step t+m
2. $\mathbf{s_t}$, smoothed value of the *Level* of time serie at step t
3. $\mathbf{b_t}$, smmothed value of the *Trend* of time serie at step t
4. $\mathbf{c_t}$, smmothed value of the *Seasonality* of time serie at step t
5. $\mathbf{L}$, period of seasonality
6. α, smoothing paramiter for the Level estimate, must lie in the [0,1] interval.
7. β, smoothing paramiter for the Trend estimate, must lie in the [0,1] interval.
8. γ, smoothing paramiter for the Seasonality estimate, must lie in the [0,1] interval.
9. $\mathbf{X_t}$, value of time serie at step t
10. $\mathbf{t}$, current time

The showed equations underline the following cues of the algorithm:

1. Holt-Winters algorithm perform a Triple Exponential Smoothing respectively on: Level, Trend and Seasonality features of the time serie analyzed.
2. Holt-Winters combine the smoothed values in order to do the forecasts
3. Seasonality smoothed value can be used in a *moltiplicative* or *additive* manner in order to do forecast

The Seasonal Additive version of the algorithm is the following:

$$\begin{cases} F_{t+m} & = s_t + m \cdot b_t + c_{t-L+m \ mod \ L} \\ s_t & = \alpha \cdot (X_t - c_{t-L}) + (1-\alpha) \cdot (s_{t-1} + b_{t-1}) \\ b_t & = \beta \cdot (s_t - s_{t-1}) + (1-\beta) \cdot b_{t-1} \\ c_t & = \gamma \cdot (X_t - s_{t-1} - b_{t-1}) + (1-\gamma) \cdot c_{t-L} \end{cases}$$

The symbols employed are the same of the Multiplicative Version.

Prophet

Prophet is a time series decomposable model that encapsulates threee important cues of time series:

1. Trend
2. Seasonality
3. Holidays

The base equation of the model is the following:

$$\begin{aligned} y(t) &= g(t) + s(t) + h(t) + \varepsilon_t \\ g(t) &= \text{Trend Model} \\ s(t) &= \text{Season Model} \\ h(t) &= \text{Holiday Model} \\ \varepsilon_t &= \text{Model Error} \end{aligned}$$

The showed model has the following properties:

1. Prophet is an Additive Model
2. Prophet cast Forecasting problem in a Curve-Fitting one
3. Prophet analytical equation help adding new seasonal effects to the model
4. Unlike SARIMA, we don't have to remove outliers

Trend Model

Trend Model of prophet is declined in two flaws based on the type of studied time serie.

The flaws of Trend Model are: *Saturating Grow Model* and *Point-Wise Linear Model*

Saturating Grow Model interprets trend variations as a Saturating Non-Linear Growth phenomenon where saturation occurs at a specific *Carrying Capacity*.

General equation of the model is the following:

$$
\begin{aligned}
g(t) &= \frac{C(t)}{1 + e^{\,-(k+a(t)^T \cdot\delta)\cdot(t\,-\,(m+a(t)^T \cdot\gamma)\,)}} \\
m &= \text{Offset Paramiter} \\
k &= \text{Base Growth Rate} \\
s &= \text{Vector of Growth Change Times} \\
\delta &= \text{Vector of Growth Change occuring at times } s_t \\
\delta &\in Laplace(0, \tau) \\
S &= \text{Total Number of Growth Changes} \\
C(t) &= \text{Carrying Capacity at time t} \\
a_j(t) &= \begin{cases} 1 & \text{if } t \geq s_j \\ 0 & \text{otherwise} \end{cases} \\
\gamma &= \text{Offset Adjusstment Params} \\
\gamma_t &= \left(s_j - m - \sum_{l<j} \gamma_l \right) \cdot \left(1 - \frac{k + \sum_{l<j} \delta_l}{k + \sum_{l \leq j} \delta_l} \right)
\end{aligned}
$$

Point-Wise Linear Model is useful when time serie don't show a Saturating Growth trend.

This model has a costant growth rate and implies the following equation:

$$
\begin{aligned}
g(t) &= (k + a(t)^T \cdot \delta) \cdot t + (m + a(t)^T \cdot \gamma) \\
\gamma_j &= -s_j \cdot \delta_j
\end{aligned}
$$

Seasonal Model

Seasonal Model use a function that is periodic of time in order to model data.

More specific, Seasonal Model use a *Standard Fourier Series*.

Seasonal Model Equation is reported below:

$$\begin{aligned} s(t) &= \sum_{n=1}^{N} a_n \cdot cos\left(\frac{2 \cdot \pi \cdot t}{P}\right) + b_n \cdot sin\left(\frac{2 \cdot \pi \cdot t}{P}\right) \\ \beta &= [a_1, b_1, \ldots, a_N, b_N]^T \\ \beta &= \text{Paramiters to estimate, Fourier Series coefficients} \\ N &= \text{Number of harmonics of Series} \\ P &= \text{Period} \end{aligned}$$

It is useful to point out that when $N \to \infty$ model capacity grows at cost of overfitting.

Holiday Model

This model incorporates Holiday effects in the forecast. This is done by using base assumption that holidays are mutually independent.

The equation of Holiday Model is the following:

$$\begin{aligned} h(t) &= Z(t) \cdot k \\ Z(t) &= [\, 1(t \in D_1), \ldots, 1(t \in D_M)\,] \\ 1(x) &= \text{1 if x = True else False} \\ D_i &= \text{list of past and future dates for holiday i} \\ K_i &= \text{Forecast influence of holday i} \\ K &\in \mathcal{N}(0, \upsilon) \\ Z(t) &\in \mathbb{R}^{M \times N} \\ K &\in \mathbb{R}^{N} \\ N &= \text{Number of Holidays} \\ M &= \text{Number of time slices analized} \end{aligned}$$

Another important feature of holiday model is the support to the effects of surrounding days of holiday; these are treated theirself as holidays.

Model Fitting & Summary

Entire Prophet model is fitted using *L-BFGS* algorithm, a version of *Quasi-Newton* methods particularly optimized in term of memory consummation.

From the point-view of analyst, model provides the following set of hyperparameters:

1. Capacity
2. Trend Change Points
3. Seasonality and Holidays Events
4. τ, smoothing Trend Model
5. (σ, υ), smoothing Seasonal and Holiday Model.

Models Evaluation

In order to evaluate forecast models, we choose the following three strategies:

1. **RollingWindow**, this kind of evaluation prescribe splitting dataset in *fixed-dimension* train and test-set.

Then, it will start an *Iterative Procedure* where at each iteration: 1) Start location of the training dataset moves forward by a fixed amount, while the test dataset “slides” forward to accommodate.

2) Fit Model on training-set of 1) and evaluate on residual Test-Set using a set of evaluation measures

Iterations continue until the end of the test set meets the end of the full data set.

2. **ExpandingWindow**, in this scenario dataset is also splitted in training and test-set but unlike *RollingWindow* dimension of these sets can vary during evaluation method.

   Method is an iterative procedure where at each iteration: 1) Size of the training dataset increases by a fixed amount, while the test dataset “slides” forward to accommodate.

   2) Fit Model on training-set of 1) and evaluate on residual Test-Set using a set of evaluation measures

   Iterations continue until the complete data set is used to either train or test in the final interation.

3. **TrainTest**, in this scenario dataset is splitted in training and test-set.

   Then, model was fitted on training-set and evaluated on test-set following a set of evaluation measures.

The first two aforementioned methods are complementary and cover two aspects of models:

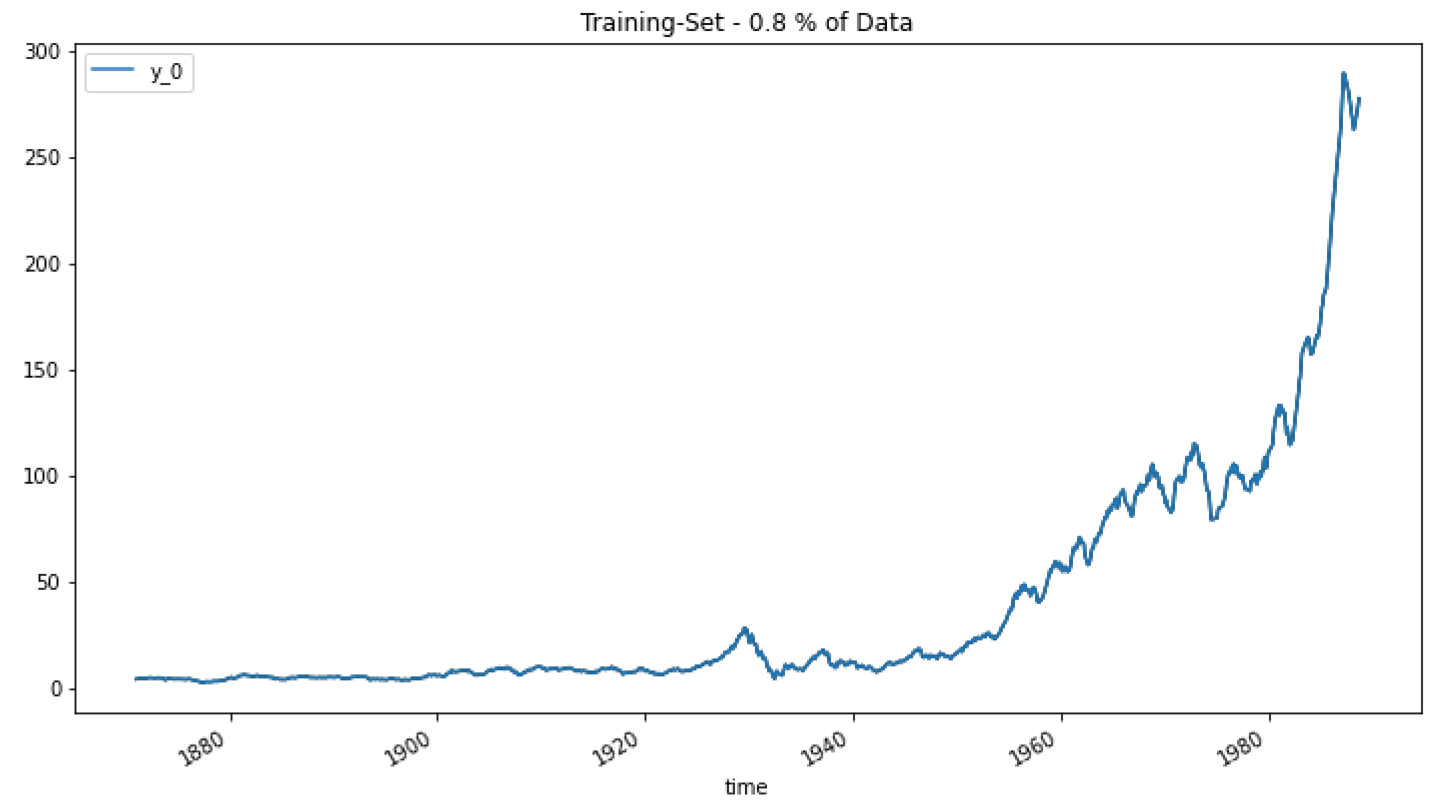
1. RollingWindow allows to understand the behavior of model at different bands of times.
2. ExpandingWindow allows to understand the behavior of model at increasing of training-set size.

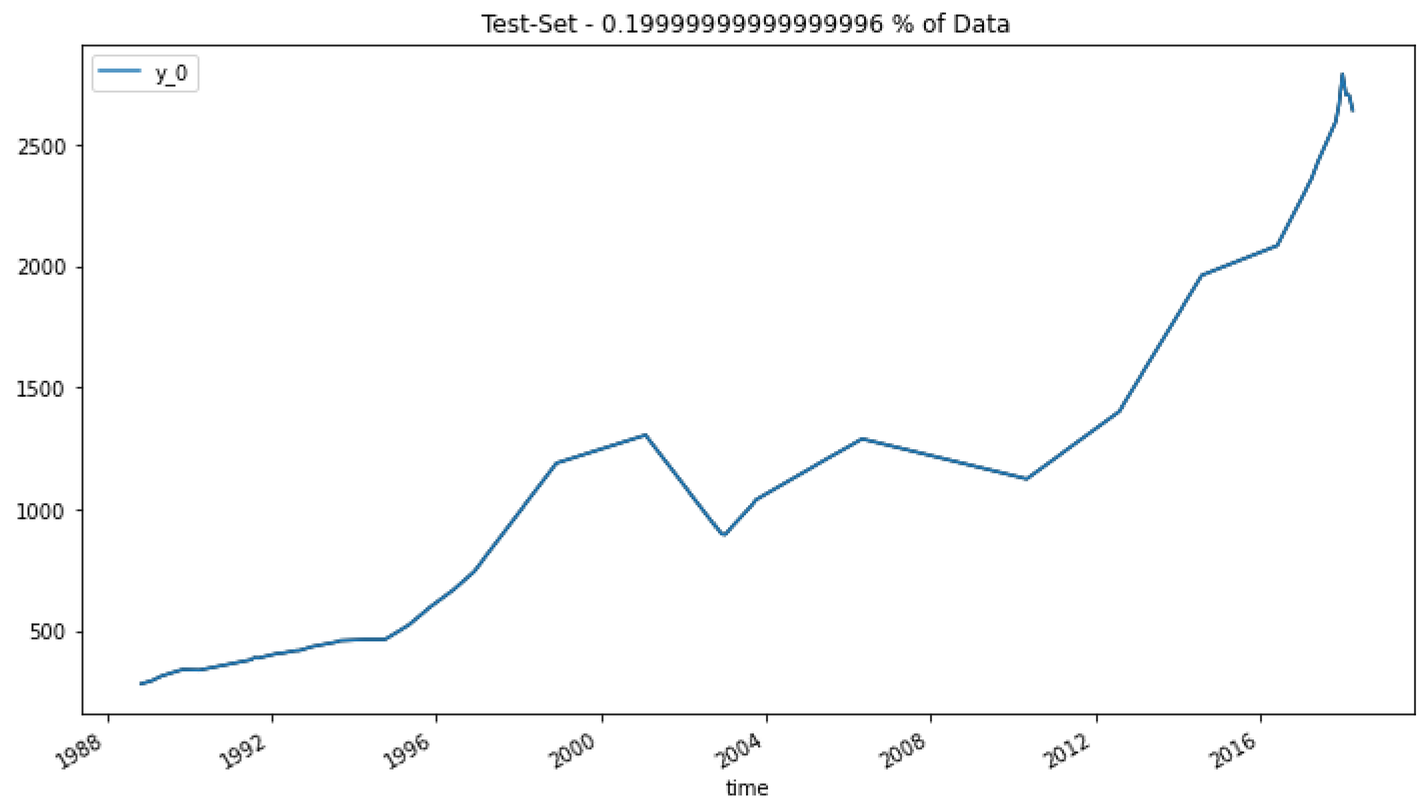
However, all methods use the following set of evaluation measures:

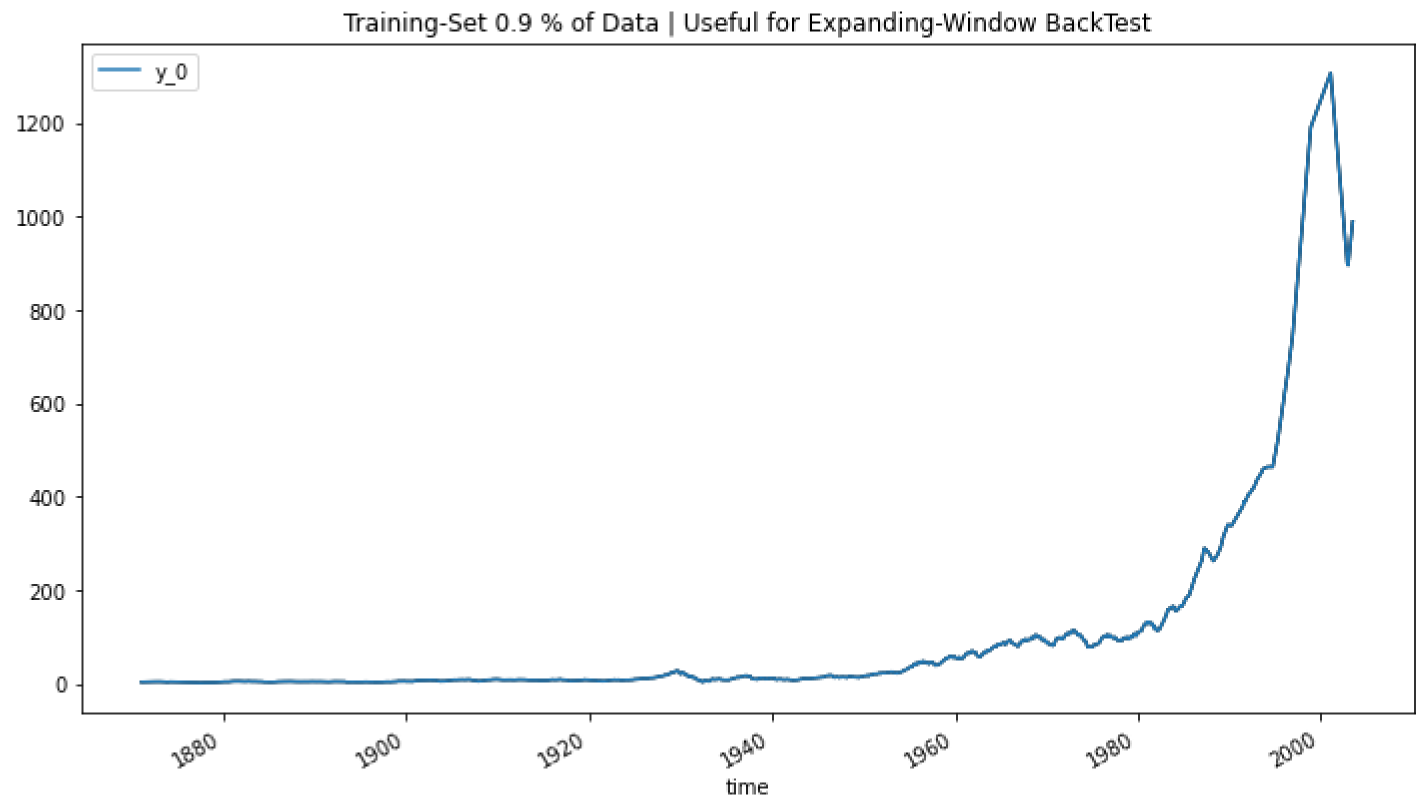
- Mean Square Error (MSE)
- Root Mean Square Error (RMSE)
- Mean Absolute Error (MAE)
- Mean Absolute Percentage Error (MAPE)

Experiments

In [34]:
```
#Split Dataset
PERC_TRAIN:float                   = 0.8
split:int                          = int(PERC_TRAIN*len(sp_cleaned))
split_90_perc:int                  = int(0.9*len(sp_cleaned))


training_set:TimeSeriesData        = sp_cleaned[:split]
test_set:TimeSeriesData            = sp_cleaned[split:]
training_set_90_perc:TimeSeriesData = sp_cleaned[:split_90_perc]


training_set.plot(cols=['y_0'])
plt.title("Training-Set - {} % of Data".format(PERC_TRAIN))

test_set.plot(cols=['y_0'])
plt.title("Test-Set - {} % of Data".format(1-PERC_TRAIN))

training_set_90_perc.plot(cols=['y_0'])
plt.title("Training-Set {} % of Data | Useful for Expanding-Window BackTest".format(0.9))


plt.show()
```







Fitting Models

All models are fitted with a GridSearch and MAE error measure.

In [35]:
```
#Fit SARIMA
def evaluation_sarima(params:dict) -> float:
    sarima_params:SARIMAParams = SARIMAParams(
        p = params['p'],
        d = params['d'],
        q = params['q'],
        seasonal_order = ( params['p_seas'] , params['d_seas'] , params['q_seas'] , 12 )
    )
    model:SARIMAModel       = SARIMAModel(training_set, sarima_params)
    model.fit()
    model_pred:DataFrame    = model.predict(steps=len(test_set))
    error:float             = np.mean( np.abs( model_pred['fcst'].values - test_set.value.values ) )
    return error
```

In [36]:
```
sarima_grid_search = [
{
    "name": "p",
    "type": "choice",
    "values": list(range(1, 3)),
    "value_type": "int",
    "is_ordered": True,
},
{
    "name": "d",
    "type": "choice",
    "values": list(range(1, 3)),
    "value_type": "int",
    "is_ordered": True,
},
{
    "name": "q",
    "type": "choice",
    "values": list(range(1, 3)),
    "value_type": "int",
    "is_ordered": True,
},
{
    "name": "p_seas",
    "type": "choice",
    "values": list(range(1, 3)),
    "value_type": "int",
    "is_ordered": True,
},
{
    "name": "d_seas",
    "type": "choice",
    "values": list(range(1, 3)),
    "value_type": "int",
    "is_ordered": True,
},
{
    "name": "q_seas",
    "type": "choice",
    "values": list(range(1, 3)),
    "value_type": "int",
    "is_ordered": True,
}
]
```

In [37]:
```python
data_files          = listdir("data")

if "sarima_eval.csv" in data_files:
    # Load Previous Evaluation from Disk
    sarima_tuning   = pd.read_csv("data/sarima_eval.csv")
else:
    # Perform online Evaluation
    sarima_tuner = tpt.SearchMethodFactory.create_search_method(
        objective_name="evaluation_metric",
        parameters=sarima_grid_search,
        selected_search_method=SearchMethodEnum.GRID_SEARCH,
    )


    sarima_tuner.generate_evaluate_new_parameter_values(evaluation_function=evaluation_sarima)

    sarima_tuning = (sarima_tuner.list_parameter_value_scores())
    sarima_tuning.to_csv("data/sarima_eval.csv")
```

In [38]:
```
idx_min:int     =  sarima_tuning['mean'].idxmin()
record:Series   =  sarima_tuning.iloc[idx_min,:]

mean_error      = record['mean']
params:dict     = record['parameters']

print("[!] Best Model Mean Error: {} \n[!] Best Model Params:\n\t-> {}".format(mean_error , params))
print("\n[!] Sarima GridSearch Params: \n")
pprint(sarima_grid_search)
```

```
[!] Best Model Mean Error: 194.8710567746957
[!] Best Model Params:
        -> {'p': 1, 'd': 2, 'q': 2, 'p_seas': 1, 'd_seas': 2, 'q_seas': 1}

[!] Sarima GridSearch Params:

[{'is_ordered': True,
  'name': 'p',
  'type': 'choice',
  'value_type': 'int',
  'values': [1, 2]},
 {'is_ordered': True,
  'name': 'd',
  'type': 'choice',
  'value_type': 'int',
  'values': [1, 2]},
 {'is_ordered': True,
  'name': 'q',
  'type': 'choice',
  'value_type': 'int',
  'values': [1, 2]},
 {'is_ordered': True,
  'name': 'p_seas',
  'type': 'choice',
  'value_type': 'int',
  'values': [1, 2]},
 {'is_ordered': True,
  'name': 'd_seas',
  'type': 'choice',
  'value_type': 'int',
  'values': [1, 2]},
 {'is_ordered': True,
  'name': 'q_seas',
  'type': 'choice',
  'value_type': 'int',
  'values': [1, 2]}]
```

```
In [39]: #Fit Holt-Winters
         def evaluate_hw(params:dict):
             hw_params:ProphetParams = HoltWintersParams(
                     seasonal=params['seasonal'],
                     trend=params['trend'],
                     seasonal_periods=12
                     )

             model:HoltWintersModel      = HoltWintersModel(training_set,hw_params)
             model.fit()
             model_pred:DataFrame        = model.predict(steps=len(test_set) )
             error:float                 = np.mean( np.abs( model_pred['fcst'].values - test_set.value.values ) )

             return error
```

```
In [40]: hw_grid_search = [
         {
             "name": "trend",
             "type": "choice",
             "values": [ 'add' , 'mul'],
             "value_type": "str",
             "is_ordered": False,
         },
         {
             "name": "seasonal",
             "type": "choice",
             "values": [ 'add' , 'mul'],
             "value_type": "str",
             "is_ordered": False,
         }
         ]
```

In [41]:

```
hw_tuner = tpt.SearchMethodFactory.create_search_method(
    objective_name="evaluation_metric",
    parameters=hw_grid_search,
    selected_search_method=SearchMethodEnum.GRID_SEARCH,
)


hw_tuner.generate_evaluate_new_parameter_values(evaluation_function=evaluate_hw)

# Retrieve parameter tuning results
hw_tuning = (hw_tuner.list_parameter_value_scores())
hw_tuning
```

```
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "trend". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "trend". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "trend". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "trend". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "trend". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:24:38] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
```

Out[41]:

	arm_name	metric_name	mean	sem	trial_index	parameters
0	0_0	evaluation_metric	456.078902	0.0	0	{'trend': 'add', 'seasonal': 'add'}
1	0_1	evaluation_metric	391.831152	0.0	0	{'trend': 'add', 'seasonal': 'mul'}
2	0_2	evaluation_metric	302.772284	0.0	0	{'trend': 'mul', 'seasonal': 'add'}
3	0_3	evaluation_metric	1208.252390	0.0	0	{'trend': 'mul', 'seasonal': 'mul'}

In [42]:
```
idx_min:int     = hw_tuning['mean'].idxmin()
record:Series   = hw_tuning.iloc[idx_min,:]

mean_error      = record['mean']
params:dict     = record['parameters']

print("[!] Best Model Mean Error: {} \n\n[!] Best Model Params:\n\t-> {}".format(mean_error , params))
print("\n\n[!] Holt-Winters GridSearch Params: \n")
pprint(hw_grid_search)
```

```
[!] Best Model Mean Error: 302.7722835707592

[!] Best Model Params:
        -> {'trend': 'mul', 'seasonal': 'add'}


[!] Holt-Winters GridSearch Params:

[{'is_ordered': False,
  'name': 'trend',
  'type': 'choice',
  'value_type': 'str',
  'values': ['add', 'mul']},
 {'is_ordered': False,
  'name': 'seasonal',
  'type': 'choice',
  'value_type': 'str',
  'values': ['add', 'mul']}]
```

In [43]:
```
#Fit Prophet
def evaluate_prophet(params:dict):
    if params['growth'] == 'logistic':
        pr_params:ProphetParams = ProphetParams(
            growth=params['growth'],
            seasonality_mode=params['seasonal'],
            cap=2800
                    )
    else:
        pr_params:ProphetParams = ProphetParams(
            growth=params['growth'],
            seasonality_mode=params['seasonal']
                    )

    model:ProphetModel          = ProphetModel(data=training_set, params=pr_params)
    model.fit()
    model_pred:DataFrame        = model.predict(steps=len(test_set) )
    error:float                 = np.mean( np.abs( model_pred['fcst'].values - test_set.value.values ) )

    return error
```

```
In [44]: prophet_grid_search = [
         {
             "name": "growth",
             "type": "choice",
             "values": [ 'linear' , 'logistic'],
             "value_type": "str",
             "is_ordered": False,
         },
         {
             "name": "seasonal",
             "type": "choice",
             "values": [ 'additive' , 'multiplicative'],
             "value_type": "str",
             "is_ordered": False,
         }
         ]
```

In [45]:
```
prophet_tuner = tpt.SearchMethodFactory.create_search_method(
    objective_name="evaluation_metric",
    parameters=prophet_grid_search,
    selected_search_method=SearchMethodEnum.GRID_SEARCH,
)


prophet_tuner.generate_evaluate_new_parameter_values(evaluation_function=evaluate_prophet)

# Retrieve parameter tuning results
prophet_tuner = (prophet_tuner.list_parameter_value_scores())
prophet_tuner
```

```
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "growth". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "growth". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "growth". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "growth". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "growth". Defaulting to `False` for parameters of `ParameterType`
STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
[WARNING 11-06 17:25:08] ax.core.parameter: `sort_values` is not specified for `ChoiceParameter` "seasonal". Defaulting to `False` for parameters of `ParameterTyp
e` STRING. To override this behavior (or avoid this warning), specify `sort_values` during `ChoiceParameter` construction.
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
```

Out[45]:

	arm_name	metric_name	mean	sem	trial_index	parameters
0	0_0	evaluation_metric	847.776606	0.0	0	{'growth': 'linear', 'seasonal': 'additive'}
1	0_1	evaluation_metric	847.824852	0.0	0	{'growth': 'linear', 'seasonal': 'multiplicati...
2	0_2	evaluation_metric	600.830344	0.0	0	{'growth': 'logistic', 'seasonal': 'additive'}
3	0_3	evaluation_metric	607.117477	0.0	0	{'growth': 'logistic', 'seasonal': 'multiplica...

In [46]:
```
idx_min:int     = prophet_tuner['mean'].idxmin()
record:Series   = prophet_tuner.iloc[idx_min,:]

mean_error      = record['mean']
params:dict     = record['parameters']

print("[!] Best Model Mean Error: {} \n\n[!] Best Model Params:\n\t-> {}".format(mean_error , params))
print("\n\n[!] Prophet GridSearch Params: \n")
pprint(prophet_grid_search)
```

```
[!] Best Model Mean Error: 600.8303436905337

[!] Best Model Params:
        -> {'growth': 'logistic', 'seasonal': 'additive'}


[!] Prophet GridSearch Params:

[{'is_ordered': False,
  'name': 'growth',
  'type': 'choice',
  'value_type': 'str',
  'values': ['linear', 'logistic']},
 {'is_ordered': False,
  'name': 'seasonal',
  'type': 'choice',
  'value_type': 'str',
  'values': ['additive', 'multiplicative']}]
```

Models Evaluation - Simple

In [47]:
```
backtester_errors = {}
```

In [48]:
```
params     = SARIMAParams(p=1, d=2, q=2 , seasonal_order=(1,2,1,12))
ALL_ERRORS:List[str] = ['mape', 'mae', 'mse', 'rmse']

backtester_sarima = BackTesterSimple(
    error_methods=ALL_ERRORS,
    data=sp_cleaned,
    params=params,
    train_percentage=80,
    test_percentage=20,
    model_class=SARIMAModel)

backtester_sarima.run_backtest()


backtester_errors['sarima'] = {}
for error, value in backtester_sarima.errors.items():
    backtester_errors['sarima'][error] = value
```

In [49]:
```
params_prophet = ProphetParams(seasonality_mode='additive' , growth='logistic', cap=2800)

backtester_prophet = BackTesterSimple(
    error_methods=ALL_ERRORS,
    data=sp_cleaned,
    params=params_prophet,
    train_percentage=80,
    test_percentage=20,
    model_class=ProphetModel)

backtester_prophet.run_backtest()

backtester_errors['prophet'] = {}
for error, value in backtester_prophet.errors.items():
    backtester_errors['prophet'][error] = value
```

```
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
```

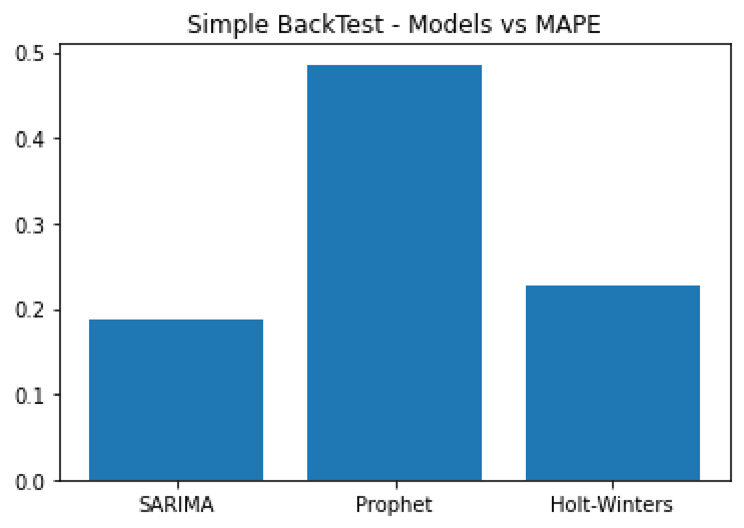
In [50]:
```
params_hw = HoltWintersParams(trend='mul' ,seasonal='additive' , seasonal_periods=12 )

backtester_prophet = BackTesterSimple(
    error_methods=ALL_ERRORS,
    data=sp_cleaned,
    params=params_hw,
    train_percentage=80,
    test_percentage=20,
    model_class=HoltWintersModel)

backtester_prophet.run_backtest()

backtester_errors['holt'] = {}
for error, value in backtester_prophet.errors.items():
    backtester_errors['holt'][error] = value
```

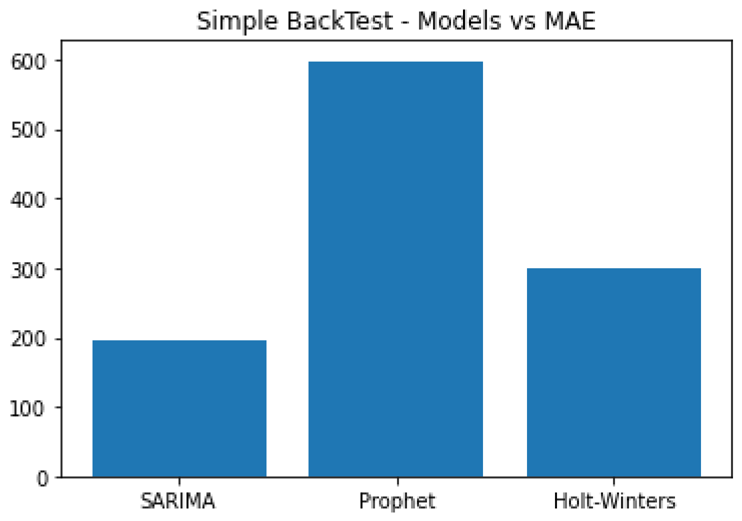
In [51]:
```
simple_backtest_outcome = pd.DataFrame.from_dict(backtester_errors)
simple_backtest_outcome
```

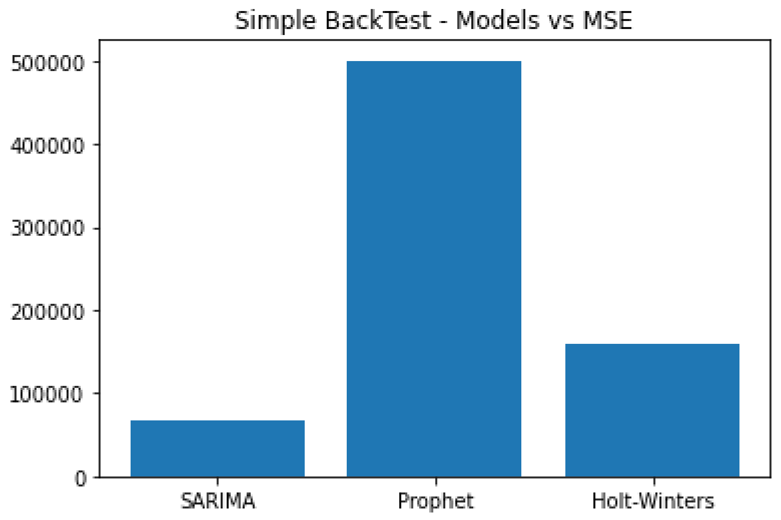
Out[51]:

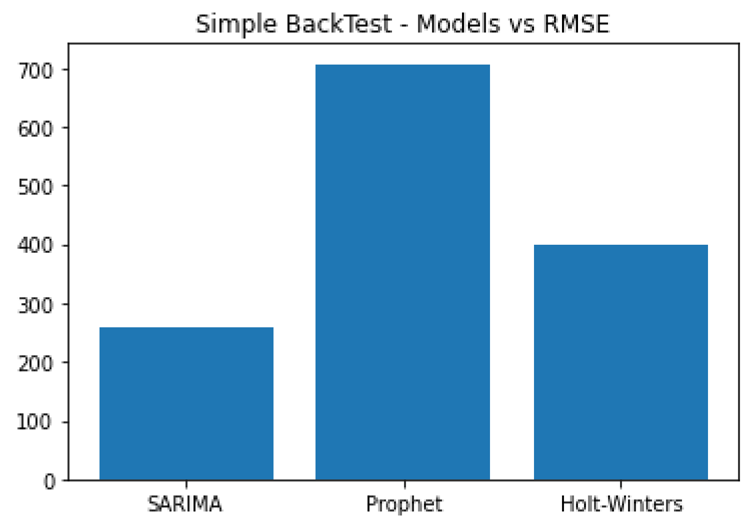
	sarima	prophet	holt
mape	0.186254	0.485578	0.226715
mae	194.464279	597.744972	300.569592
mse	67556.226050	499938.338276	158052.014521
rmse	259.915806	707.063178	397.557561

```
In [52]: #Plot Barplot for Simple Backtest
simple_backtest_outcome_numpy = simple_backtest_outcome.to_numpy()

plt.bar( ["SARIMA" , "Prophet" , "Holt-Winters"] , simple_backtest_outcome_numpy[0,:] )
plt.title("Simple BackTest - Models vs MAPE")
plt.show()

plt.bar( ["SARIMA" , "Prophet" , "Holt-Winters"] , simple_backtest_outcome_numpy[1,:] )
plt.title("Simple BackTest - Models vs MAE")
plt.show()

plt.bar( ["SARIMA" , "Prophet" , "Holt-Winters"] , simple_backtest_outcome_numpy[2,:] )
plt.title("Simple BackTest - Models vs MSE")
plt.show()

plt.bar( ["SARIMA" , "Prophet" , "Holt-Winters"] , simple_backtest_outcome_numpy[3,:] )
plt.title("Simple BackTest - Models vs RMSE")
plt.show()
```




Simple BackTest - Models vs MAE
0
100
200
300
400
500
600
SARIMA
Prophet
Holt-Winters


Simple BackTest - Models vs MSE
0
100000
200000
300000
400000
500000
SARIMA
Prophet
Holt-Winters



Simple Backtest underlines superiority of SARIMA over other models.

Model Evaluation - RollingWindow

In [ ]:
```
rolling_errors:dict = {}
```

In [ ]:
```
params                  = ProphetParams(seasonality_mode='additive' , growth='logistic', cap=2800)
ALL_ERRORS:List[str]    = ['mape', 'mae', 'mse', 'rmse']

rolling_prophet = BackTesterRollingWindow(
    error_methods=ALL_ERRORS,
    data=sp_cleaned,
    params=params,
    train_percentage=80,
    test_percentage=20,
    model_class=ProphetModel,
    sliding_steps=2)

rolling_prophet.run_backtest()


rolling_errors['prophet'] = {}
for error, value in rolling_prophet.errors.items():
    rolling_errors['prophet'][error] = value
```

In [ ]:
```
params      = SARIMAParams(p=1, d=2, q=2 , seasonal_order=(1,2,1,12))
ALL_ERRORS:List[str] = ['mape', 'mae', 'mse', 'rmse']

rolling_sarima = BackTesterRollingWindow(
    error_methods=ALL_ERRORS,
    data=sp_cleaned,
    params=params,
    train_percentage=80,
    test_percentage=20,
    model_class=SARIMAModel,
    sliding_steps=2,
    multi=True)

rolling_sarima.run_backtest()


rolling_errors['sarima'] = {}
for error, value in rolling_sarima.errors.items():
    rolling_errors['sarima'][error] = value
```

In [ ]:
```
params = HoltWintersParams(trend='mul' ,seasonal='additive' , seasonal_periods=12 )
ALL_ERRORS:List[str] = ['mape', 'mae', 'mse', 'rmse']

rolling_hw = BackTesterRollingWindow(
    error_methods=ALL_ERRORS,
    data=sp_cleaned,
    params=params,
    train_percentage=80,
    test_percentage=20,
    model_class=HoltWintersModel,
    sliding_steps=2)

rolling_hw.run_backtest()


rolling_errors['holt'] = {}
for error, value in rolling_hw.errors.items():
    rolling_errors['holt'][error] = value
```

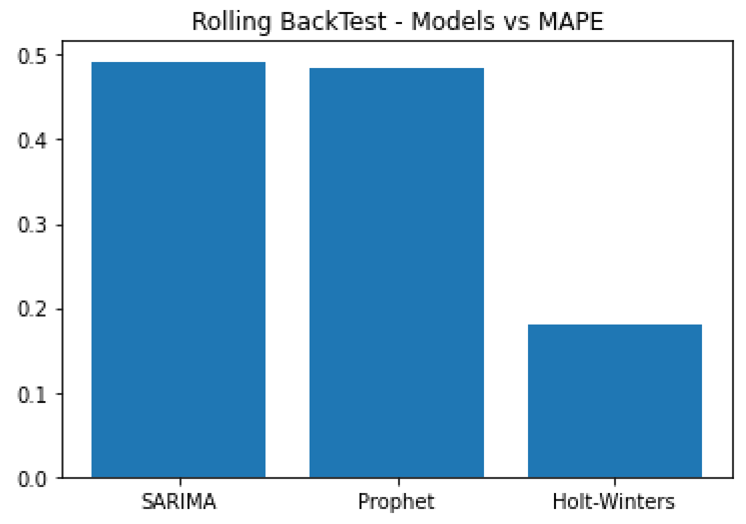
In [53]:
```
#Load Previous Evaluation Result for saving computational power
with open("data/rolling_errors.pkl", "rb") as f:
    rolling_errors_shufled = pickle.load(f)

rolling_errors              = {}
rolling_errors['sarima']    = rolling_errors_shufled['SARIMA']
rolling_errors['prophet']   = rolling_errors_shufled['prophet']
rolling_errors['holt']      = rolling_errors_shufled['holt']
```

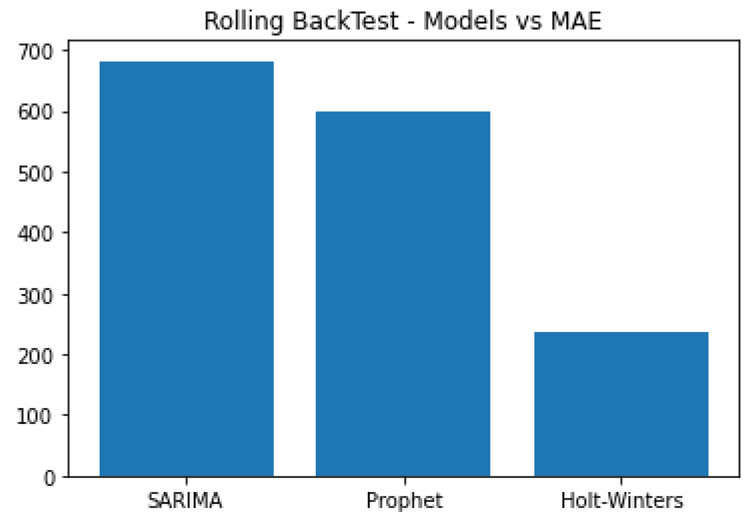
In [54]:
```
rolling_backtest_outcome = pd.DataFrame.from_dict(rolling_errors)
rolling_backtest_outcome
```

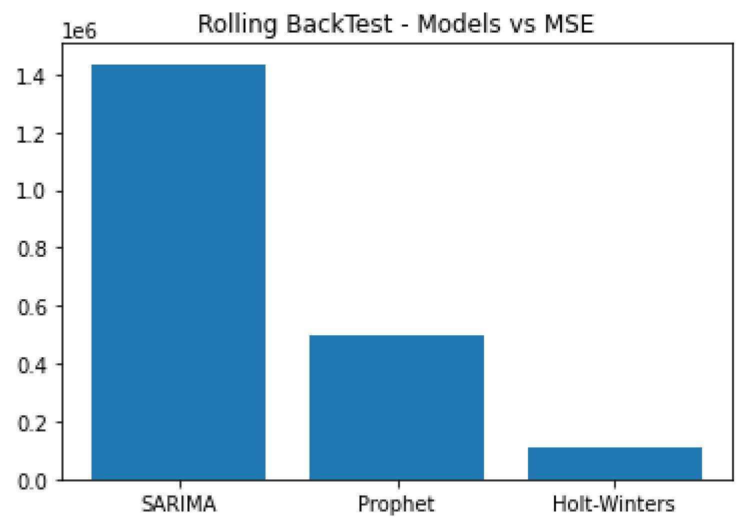
Out[54]:

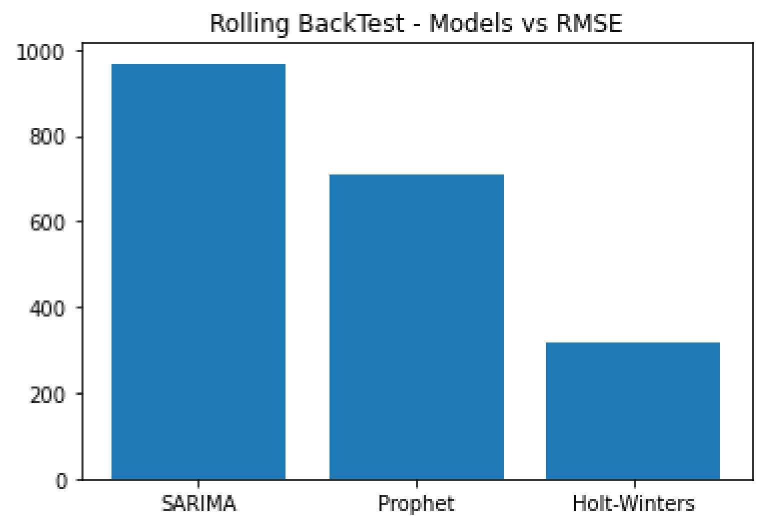
	sarima	prophet	holt
MAPE	4.920385e-01	0.484758	0.180284
MAE	6.816995e+02	598.433972	236.814092
MSE	1.436516e+06	501522.203179	107827.740297
RMSE	9.675009e+02	708.181441	318.782391

```
In [55]: #Plotting Errors for Rolling Backtest
         rolling_backtest_outcome_numpy = rolling_backtest_outcome.to_numpy()

         plt.bar( ["SARIMA" , "Prophet" , "Holt-Winters"] , rolling_backtest_outcome_numpy[0,:] )
         plt.title("Rolling BackTest - Models vs MAPE")
         plt.show()

         plt.bar( ["SARIMA" , "Prophet" , "Holt-Winters"] , rolling_backtest_outcome_numpy[1,:] )
         plt.title("Rolling BackTest - Models vs MAE")
         plt.show()

         plt.bar( ["SARIMA" , "Prophet" , "Holt-Winters"] , rolling_backtest_outcome_numpy[2,:] )
         plt.title("Rolling BackTest - Models vs MSE")
         plt.show()

         plt.bar( ["SARIMA" , "Prophet" , "Holt-Winters"] , rolling_backtest_outcome_numpy[3,:] )
         plt.title("Rolling BackTest - Models vs RMSE")
         plt.show()
```









RollingWindow BackTest underlines superiority of Holt-Winters over other models.

Models Evaluation - Expanding Window

```
In [ ]: exp_errors:dict = {}
In [ ]: params     = SARIMAParams(p=1, d=2, q=2 , seasonal_order=(1,2,1,12))
        ALL_ERRORS:List[str] = ['mape', 'mae', 'mse', 'rmse']

        exp_sarima = BackTesterExpandingWindow(
            error_methods=ALL_ERRORS,
            data=sp_cleaned,
            params=params,
            start_train_percentage=80,
            end_train_percentage=90,
            test_percentage=10,
            model_class=SARIMAModel,
            expanding_steps=2,
            multi=True)

        exp_sarima.run_backtest()


        exp_errors['sarima'] = {}
        for error, value in exp_sarima.errors.items():
            exp_errors['sarima'][error] = value
```

```
In [ ]: params = HoltWintersParams(trend='mul' ,seasonal='additive' , seasonal_periods=12 )
        ALL_ERRORS:List[str] = ['mape', 'mae', 'mse', 'rmse']

        exp_holt = BackTesterExpandingWindow(
            error_methods=ALL_ERRORS,
            data=sp_cleaned,
            params=params,
            start_train_percentage=80,
            end_train_percentage=90,
            test_percentage=10,
            model_class=HoltWintersModel,
            expanding_steps=2)

        exp_holt.run_backtest()


        exp_errors['holt'] = {}
        for error, value in exp_holt.errors.items():
            exp_errors['holt'][error] = value
```

In [ ]:
```
params = ProphetParams(seasonality_mode='additive' , growth='logistic', cap=2800)
ALL_ERRORS:List[str] = ['mape', 'mae', 'mse', 'rmse']

exp_prophet = BackTesterExpandingWindow(
    error_methods=ALL_ERRORS,
    data=sp_cleaned,
    params=params,
    start_train_percentage=80,
    end_train_percentage=90,
    test_percentage=10,
    model_class=ProphetModel,
    expanding_steps=2)

exp_prophet.run_backtest()


exp_errors['prophet'] = {}
for error, value in exp_prophet.errors.items():
    exp_errors['prophet'][error] = value
```

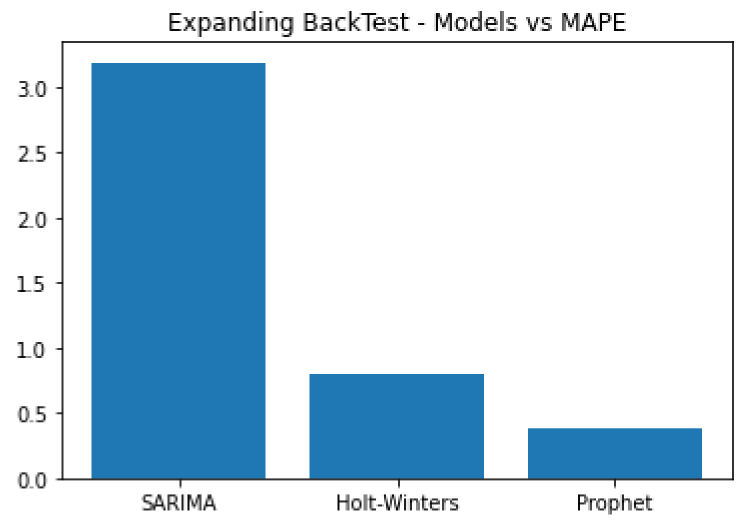
In [56]:
```
#Load Previous Evaluation Result for saving computational power
with open("data/expand_errors.pkl", "rb") as f:
    exp_errors_shuffled = pickle.load(f)

exp_errors = {}
exp_errors['sarima']    = exp_errors_shuffled['SARIMA']
exp_errors['holt']      = exp_errors_shuffled['holt']
exp_errors['prophet']   = exp_errors_shuffled['prophet']
```

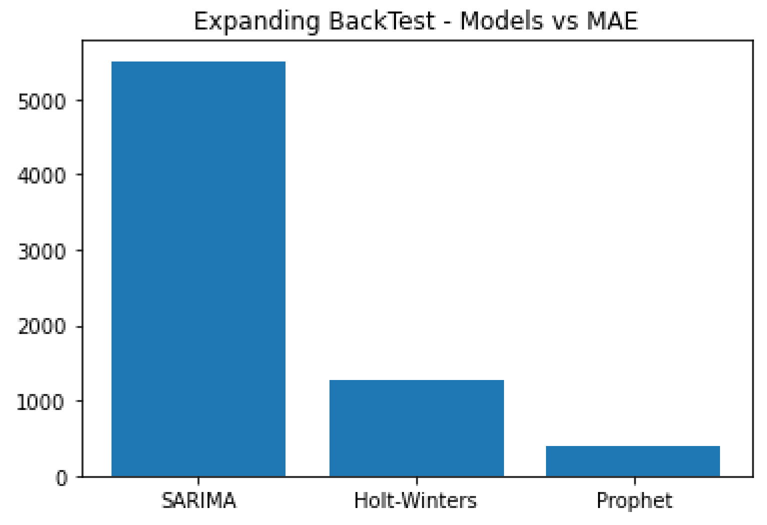
In [57]:
```
exp_backtest_outcome = pd.DataFrame.from_dict(exp_errors)
exp_backtest_outcome
```

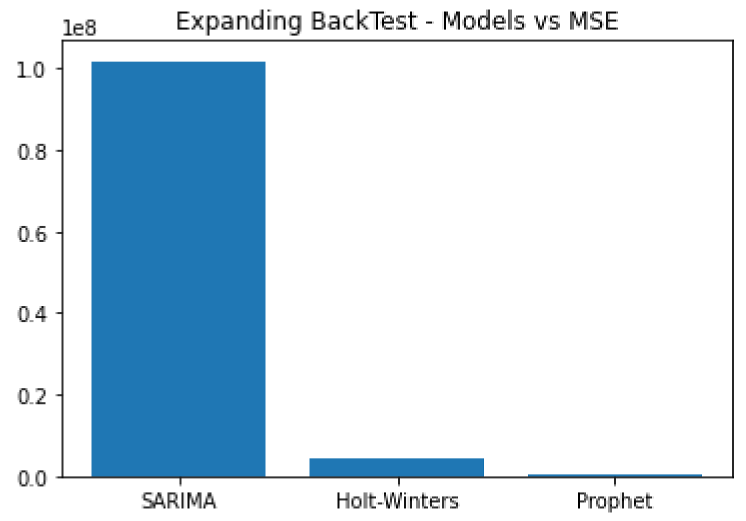
Out[57]:

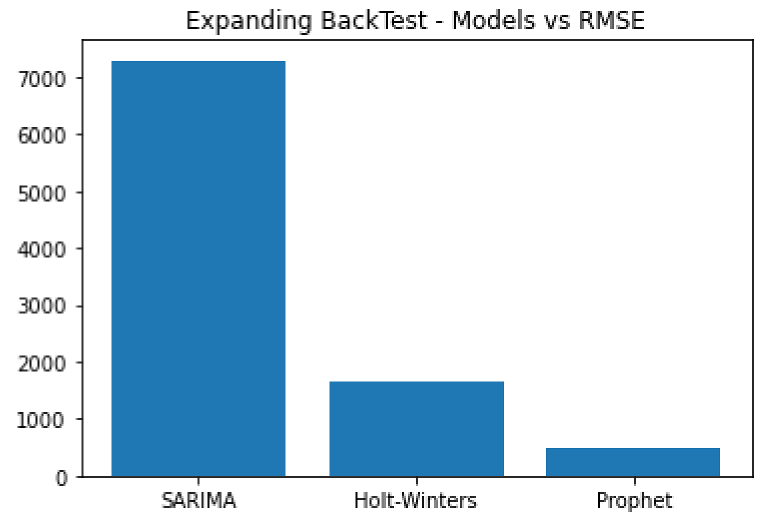
	sarima	holt	prophet
MAPE	3.184994e+00	7.901435e-01	0.384089
MAE	5.507946e+03	1.275145e+03	401.906408
MSE	1.018554e+08	4.446932e+06	223777.951361
RMSE	7.293410e+03	1.642683e+03	473.026037

```
In [89]: #Plotting Barplot for Expanding Window Backtest
         exp_backtest_outcome_numpy = exp_backtest_outcome.to_numpy()

         plt.bar( ["SARIMA" , "Holt-Winters" , "Prophet"  ] , exp_backtest_outcome_numpy[0,:] )
         plt.title("Expanding BackTest - Models vs MAPE")
         plt.show()

         plt.bar( ["SARIMA" , "Holt-Winters" , "Prophet"  ] , exp_backtest_outcome_numpy[1,:] )
         plt.title("Expanding BackTest - Models vs MAE")
         plt.show()

         plt.bar( ["SARIMA" , "Holt-Winters" , "Prophet"  ] , exp_backtest_outcome_numpy[2,:] )
         plt.title("Expanding BackTest - Models vs MSE")
         plt.show()

         plt.bar( ["SARIMA" , "Holt-Winters" , "Prophet"  ] , exp_backtest_outcome_numpy[3,:] )
         plt.title("Expanding BackTest - Models vs RMSE")
         plt.show()
```









ExpandingWindow BackTest underlines superiority of Prophet over other models.

Classification Experiments

In [59]:
```
def plot2d(dataset:ndarray , labels:ndarray) -> None:
    """
    # **plot2d**

    Plot clustered dataset

    Args:
        dataset         (ndarray): dataset
        labels          (ndarray): labels
    """
    data = pd.DataFrame()
    data['X'] = dataset.mean(axis=1)
    data['Y'] = dataset.std(axis=1)
    data['C'] = labels
    classes = sorted(data.C.unique())
    for c in classes:
        plt.plot(data.where(data.C==c).dropna().X.values,
                 data.where(data.C==c).dropna().Y.values,'o', markersize=10, label=str(c))
```

In [60]:
```python
def evaluate_classifier(pCls:object , pSamples:ndarray , pLabels:ndarray , pClsId:str , pMultiClass:bool=False) -> None:
    """
    # **evaluate_classifier**

    Evaluate Classifier with Confusion Matrix, Precision and Recall.

    Args:
        pCls                            (object)     : Sklearn ML Classifier
        pSamples                        (ndarray)    : test-set
        pLabels                         (ndarray)    : test-set labels
        pClsId                          (str)        : Classifier ID
        pMultiClass                     (pMultiClass | DEF = False) : True for Multi-Class Classification
    """
    print("\n\n<<<<<<<<<<< {} Classifier Evaluation >>>>>>>>>>>>".format(pClsId))

    # [1] Plot Confusion Matrix
    print("\n[!] Confusion Matrix: ")
    inference:ndarray                  = pCls.predict(pSamples)
    cmtr:ndarray                       = confusion_matrix( pLabels , inference )
    disp:ConfusionMatrixDisplay        = ConfusionMatrixDisplay( cmtr )
    disp.plot()
    plt.show()

    # [2] Print Classifier Score
    if pMultiClass:
        print("[!] Accuracy Score: {}".format(accuracy_score(pLabels , inference)))
        print("[!] Precision Score: {}".format(precision_score( pLabels , inference , average='micro')))
        print("[!] Recall Score: {}".format(recall_score( pLabels , inference, average='micro')) )
    else:
        print("[!] Accuracy Score: {}".format(accuracy_score(pLabels , inference)))
        print("[!] Precision Score: {}".format(precision_score( pLabels , inference)))
        print("[!] Recall Score: {}".format(recall_score( pLabels , inference)) )
```

SP500 Classification

This experiment consist on building a supervised learning dataset from original time series one.

In particolar we build a dataset splitting SP500 series per quarter.

In order to build labels for classification, we use two clustering algorithms:

1. Kmeans
2. Mix of Gaussians

Then, we train ,respectively for labels set, the following classifiers:

1. SVM
2. Naive Bayes
3. Logistic Regressor
4. Decision Tree
5. Random Forest

on 75 % of data and evaluate them on the ramaining 25 %.

Evaluation measures are:

- Precision
- Recall
- Accuracy
- Confusion Matrix

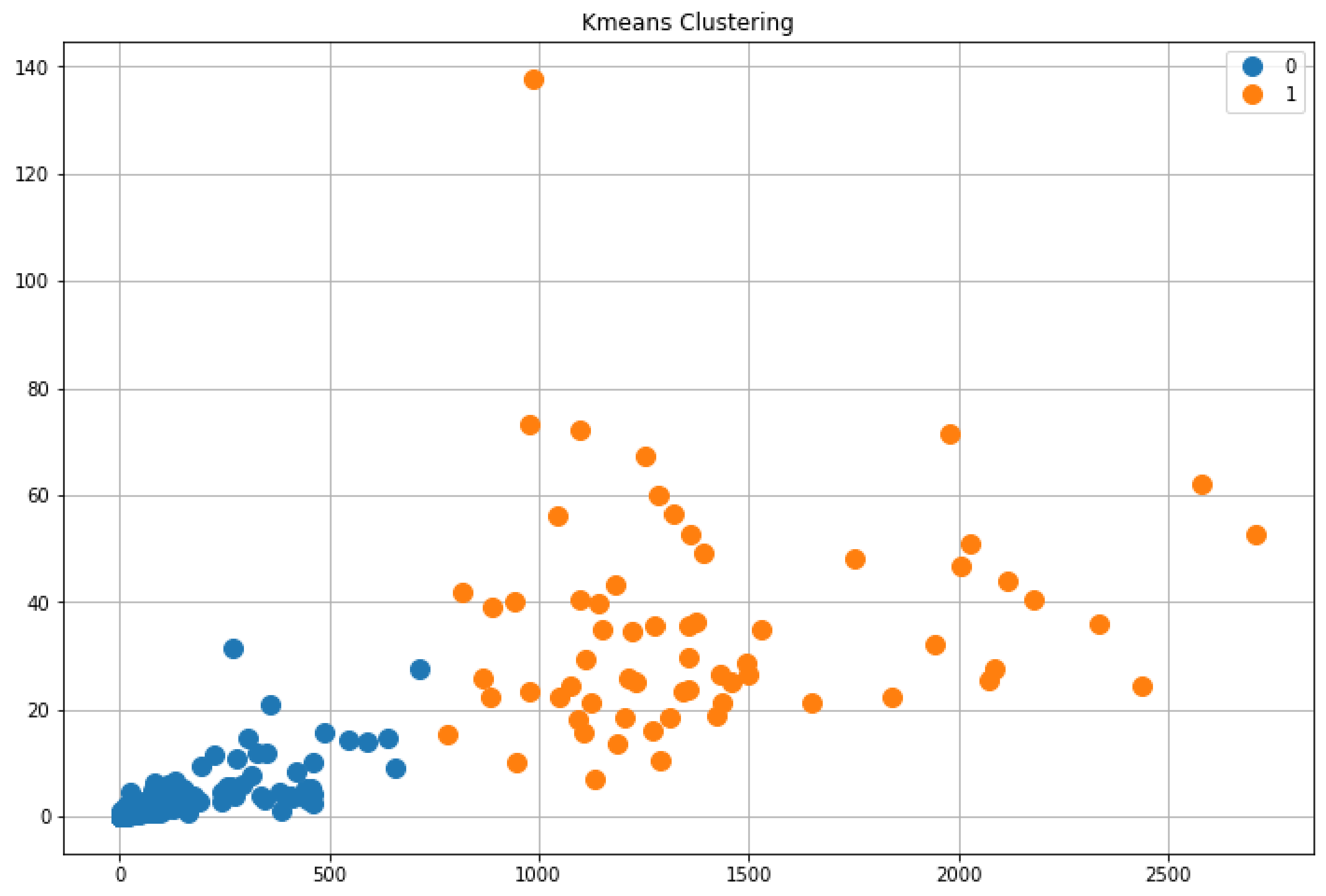
```
In [61]: #Reshape Data per Quarter
         quarter_data:ndarray = sp_numpy.reshape(-1,4)
         quarter_data.shape
```

```
Out[61]: (442, 4)
```

Kmeans

In [62]:
```
#Fit Kmeans to data
kmeans = KMeans(n_clusters=2)
kmeans.fit(quarter_data)


#Evaluate Clusters
kmeans_clusters         = kmeans.labels_.tolist()

labels_0                = [ 1 if label == 0 else 0 for label in kmeans_clusters  ]
labels_1                = [ 1 if label == 1 else 0 for label in kmeans_clusters  ]


ct_labels_0 = reduce( lambda x,y : x+y, labels_0 )
ct_labels_1 = reduce( lambda x,y : x+y, labels_1 )

print("[!] Kmeans Clustering - N Sample Class 0: {}".format(ct_labels_0))
print("[!] Kmeans Clustering - N Sample Class 1: {}".format(ct_labels_1))


#Print Clustering Stats
print("\n\n\n[!] Kmeans Clustering - Cluster Centers: \n\n{}".format(kmeans.cluster_centers_))


# Plot Clusters
plt.figure(figsize=(12,8))
plt.title("Kmeans Clustering")
plot2d(quarter_data,kmeans_clusters)
plt.legend()
plt.grid()
plt.show()
```

```
[!] Kmeans Clustering - N Sample Class 0: 378
[!] Kmeans Clustering - N Sample Class 1: 64



[!] Kmeans Clustering - Cluster Centers:

[[  63.17367725   63.67939153   64.03849206   64.59013228]
 [1397.5159375  1396.67734375 1412.58203125 1421.8834375 ]]
```

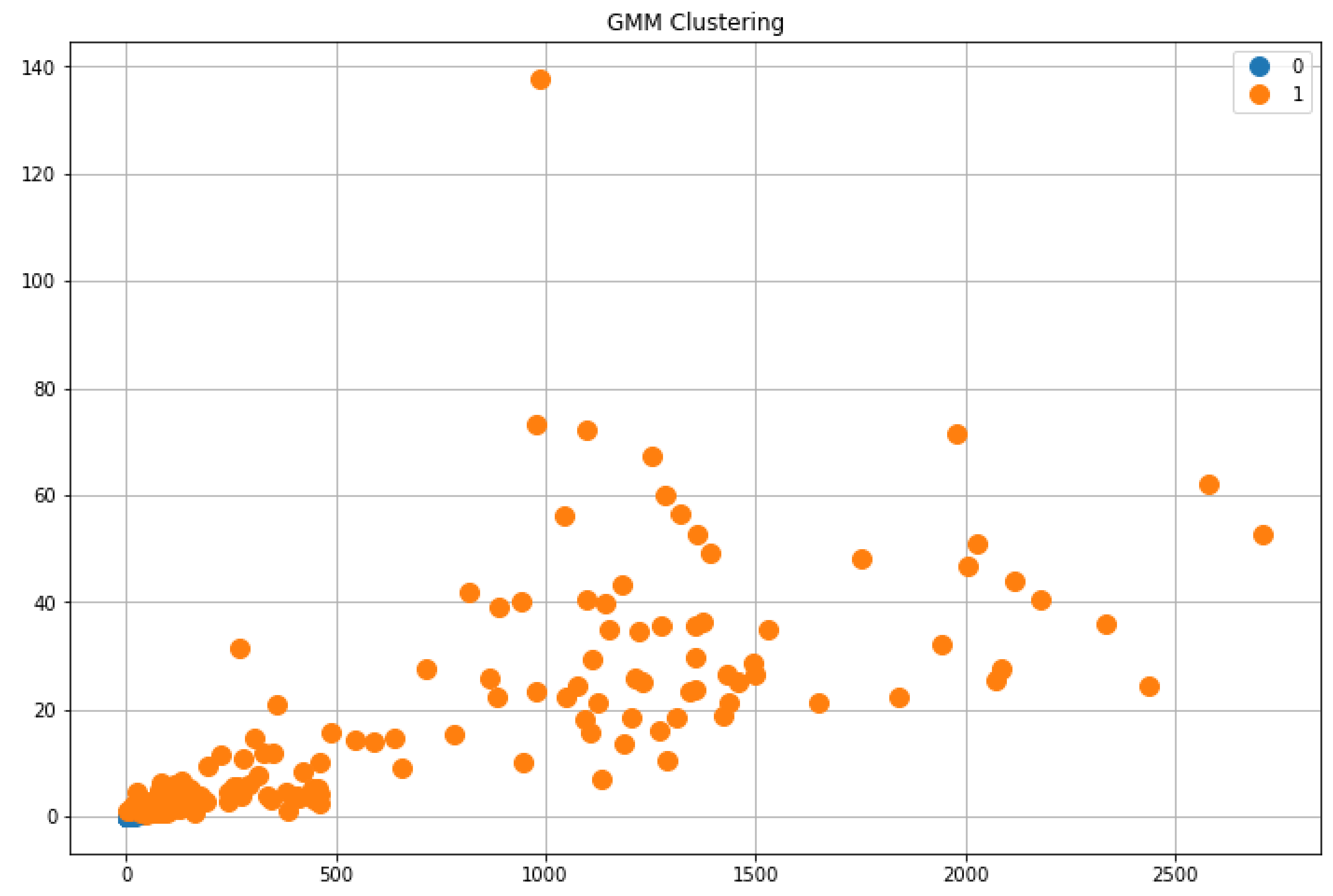


Kmeans algorithm return a clustering well-separated as underlined from graph.

Mix of Gaussians

```
In [63]: #Fit GMM to data
         gmm = GMM(n_components=2)
         gmm.fit(quarter_data)


         #Print GMM Stats
         print("Means:\n",gmm.means_)
         print("\n")
         print("Covariances:\n",gmm.covariances_)
         print("\n")
         print("Coefficients:\n",gmm.weights_)
         print("\n\n")

         #Compute Clusters
         gmm_labels = gmm.predict(quarter_data)

         labels_0 = [ 1 if label == 0 else 0 for label in gmm_labels  ]
         labels_1 = [ 1 if label == 1 else 0 for label in gmm_labels  ]

         ct_labels_0 = reduce( lambda x,y : x+y, labels_0 )
         ct_labels_1 = reduce( lambda x,y : x+y, labels_1 )

         print("[!] GMM Clustering - N Sample Class 0: {}".format(ct_labels_0))
         print("[!] GMM Clustering - N Sample Class 1: {}".format(ct_labels_1))


         #Plot Clusters
         plt.figure(figsize=(12,8))
         plt.title("GMM Clustering")
         plot2d(quarter_data,gmm_labels)
         plt.legend()
         plt.grid()
         plt.show()
```

```
Means:
 [[  9.50039865   9.53281723   9.60349074   9.64614315]
 [566.06754349 566.72813676 572.52355088 576.56982911]]


Covariances:
 [[[3.22410481e+01 3.25997019e+01 3.32325518e+01 3.37321995e+01]
  [3.25997019e+01 3.31027156e+01 3.37618826e+01 3.42528849e+01]
  [3.32325518e+01 3.37618826e+01 3.45704726e+01 3.50979076e+01]
  [3.37321995e+01 3.42528849e+01 3.50979076e+01 3.57966018e+01]]

 [[4.12644741e+05 4.12777803e+05 4.17948750e+05 4.19676219e+05]
  [4.12777803e+05 4.13767965e+05 4.19182403e+05 4.20996311e+05]
  [4.17948750e+05 4.19182403e+05 4.25433020e+05 4.27287619e+05]
  [4.19676219e+05 4.20996311e+05 4.27287619e+05 4.29701219e+05]]]


Coefficients:
 [0.55642152 0.44357848]



[!] GMM Clustering - N Sample Class 0: 246
[!] GMM Clustering - N Sample Class 1: 196
```



Mix of Gaussians return more noisy clusters than Kmeans, it suggest that classification task should be harder in Mix of Gaussians case.

Classification

Data Preparation

In [64]:
```
#Split Train-Test to 75:25
X_train_km, X_test_km, y_train_km, y_test_km = train_test_split(quarter_data,kmeans_clusters,shuffle=True ,random_state=123)
X_train_gm, X_test_gm, y_train_gm, y_test_gm = train_test_split(quarter_data,gmm_labels,shuffle=True ,random_state=123)
```

Training - Kmeans Clustering

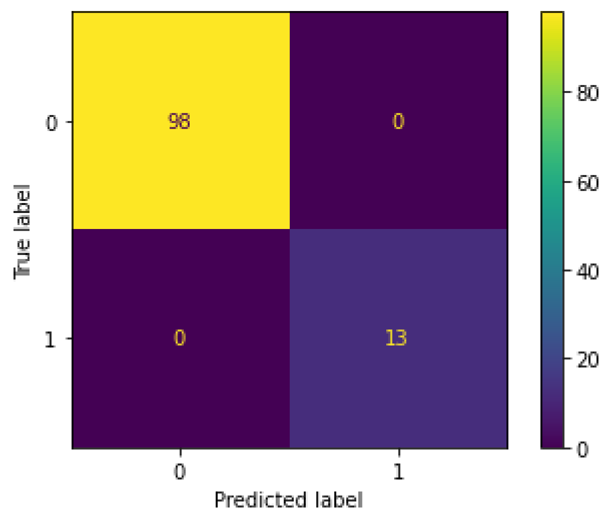
In [65]:
```
#Train Models
linear_svm:SVC                = SVC(kernel= "linear" , C=1 , max_iter=100)
linear_svm.fit(X_train_km,y_train_km)

gaussian_svm:SVC              = SVC(kernel= "rbf" , C=1 , max_iter=100)
gaussian_svm.fit(X_train_km,y_train_km)


naive_bayes                   = GaussianNB()
naive_bayes.fit(X_train_km,y_train_km)


lg_regressor                  = LogisticRegression( max_iter=100 )
lg_regressor.fit(X_train_km,y_train_km)


dec_tree                      = DecisionTreeClassifier( max_depth=2 )
dec_tree.fit(X_train_km,y_train_km)


random_forest                 = RandomForestClassifier(max_depth=2, n_estimators=2)
random_forest.fit(X_train_km,y_train_km)
```

Out[65]:
```
RandomForestClassifier(max_depth=2, n_estimators=2)
```

Evaluation - Kmeans Clustering

```
In [66]: evaluate_classifier(linear_svm,X_test_km, y_test_km, "SVM-Linear")
```

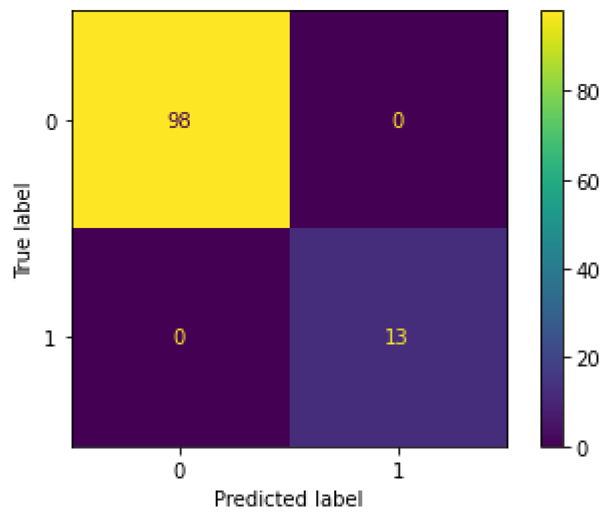
```
<<<<<<<<<< SVM-Linear Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 1.0
[!] Precision Score: 1.0
[!] Recall Score: 1.0
```

```
In [67]: evaluate_classifier(gaussian_svm,X_test_km, y_test_km, "SVM-Gaussian")
```

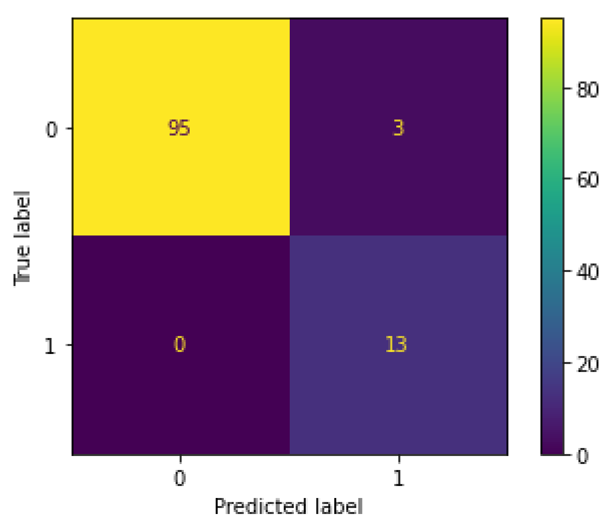
```
<<<<<<<<<< SVM-Gaussian Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 1.0
[!] Precision Score: 1.0
[!] Recall Score: 1.0
```

```
In [68]: evaluate_classifier(naive_bayes,X_test_km, y_test_km, "Naive Bayes")
```

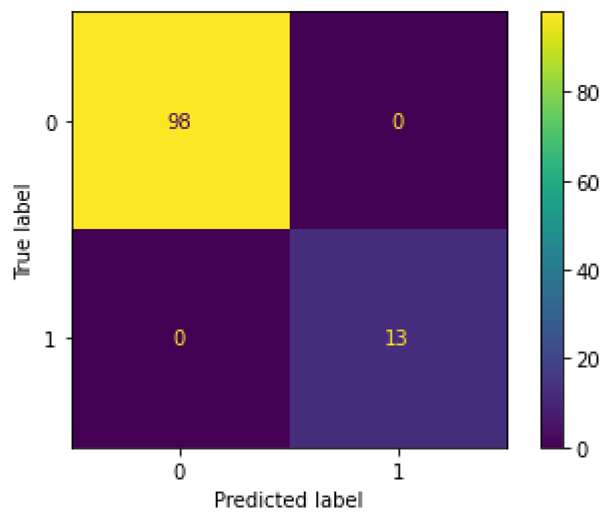
```
<<<<<<<<<< Naive Bayes Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.972972972972973
[!] Precision Score: 0.8125
[!] Recall Score: 1.0
```

```
In [69]: evaluate_classifier(lg_regressor,X_test_km, y_test_km, "Logistic Regressor")
```

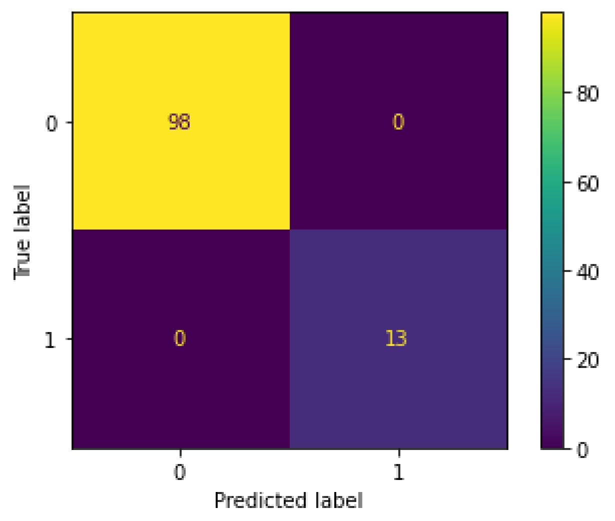
```
<<<<<<<<<< Logistic Regressor Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 1.0
[!] Precision Score: 1.0
[!] Recall Score: 1.0
```

```
In [70]: evaluate_classifier(dec_tree,X_test_km, y_test_km, "Decision Tree")
```

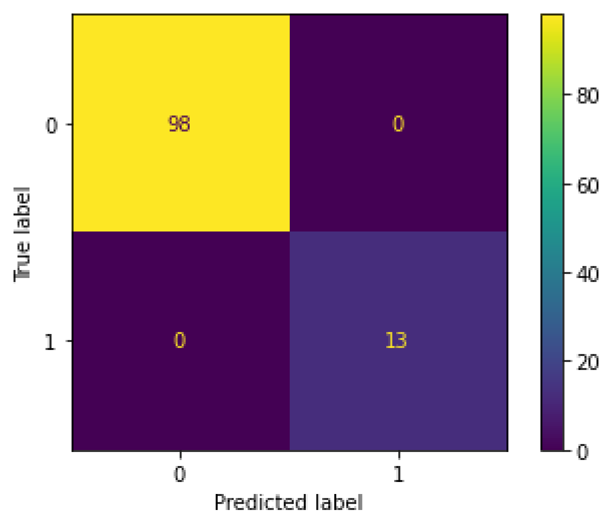
```
<<<<<<<<<< Decision Tree Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 1.0
[!] Precision Score: 1.0
[!] Recall Score: 1.0
```

```
In [71]: evaluate_classifier(random_forest,X_test_km, y_test_km, "Random Forest")
```

```
<<<<<<<<<< Random Forest Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 1.0
[!] Precision Score: 1.0
[!] Recall Score: 1.0
```

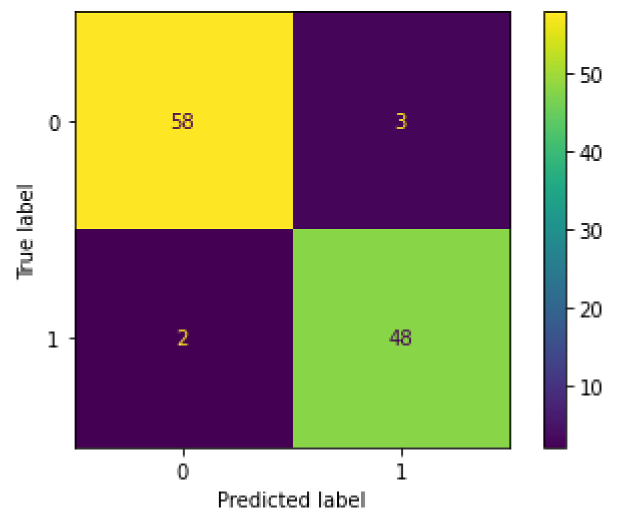
Training - GMM Labels

```
In [72]: linear_svm.fit(X_train_gm,y_train_gm)
         gaussian_svm.fit(X_train_gm,y_train_gm)
         naive_bayes.fit(X_train_gm,y_train_gm)
         lg_regressor.fit(X_train_gm,y_train_gm)
         dec_tree.fit(X_train_gm,y_train_gm)
         random_forest.fit(X_train_gm,y_train_gm)

Out[72]: RandomForestClassifier(max_depth=2, n_estimators=2)
```

Evaluation - GMM Labels

```
In [73]: evaluate_classifier(linear_svm,X_test_gm, y_test_gm, "SVM-Linear")
```

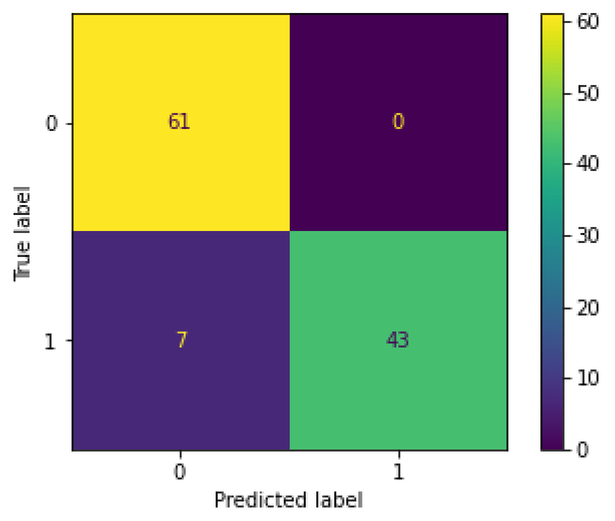
```
<<<<<<<<<< SVM-Linear Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.954954954954955
[!] Precision Score: 0.9411764705882353
[!] Recall Score: 0.96
```

```
In [74]: evaluate_classifier(gaussian_svm,X_test_gm, y_test_gm, "SVM-Gaussian")
```

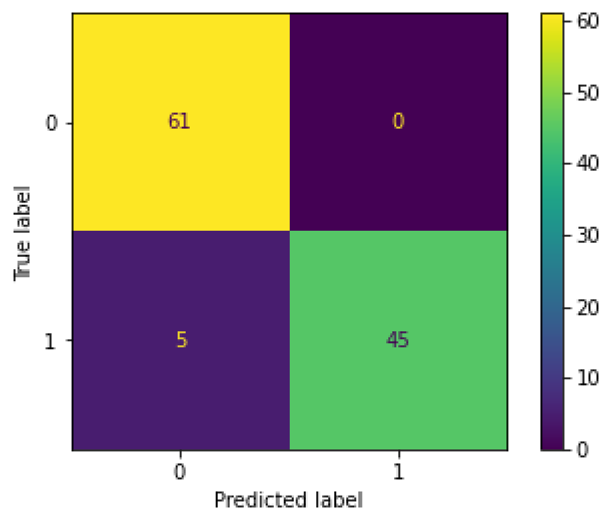
```
<<<<<<<<<< SVM-Gaussian Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.9369369369369369
[!] Precision Score: 1.0
[!] Recall Score: 0.86
```

```
In [75]: evaluate_classifier(naive_bayes,X_test_gm, y_test_gm, "Naive Bayes")
```

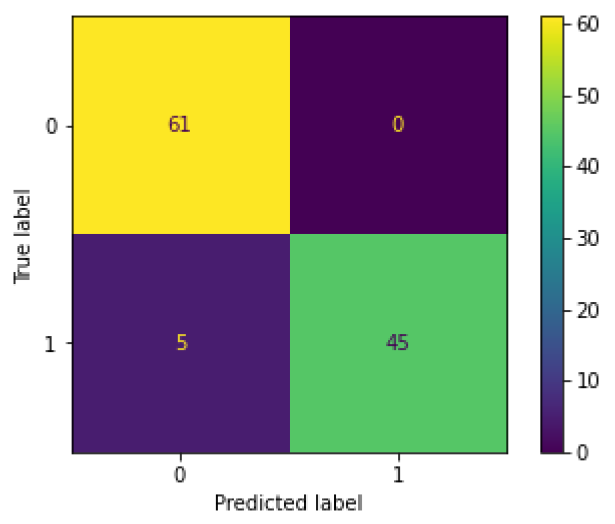
```
<<<<<<<<<< Naive Bayes Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.954954954954955
[!] Precision Score: 1.0
[!] Recall Score: 0.9
```

```
In [76]: evaluate_classifier(lg_regressor,X_test_gm, y_test_gm, "Logistic Regressor")
```

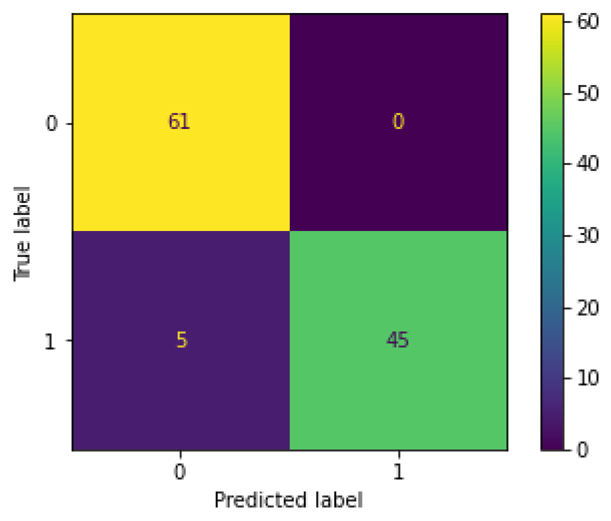
```
<<<<<<<<<< Logistic Regressor Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.954954954954955
[!] Precision Score: 1.0
[!] Recall Score: 0.9
```

```
In [77]: evaluate_classifier(dec_tree,X_test_gm, y_test_gm, "Decision Tree")
```

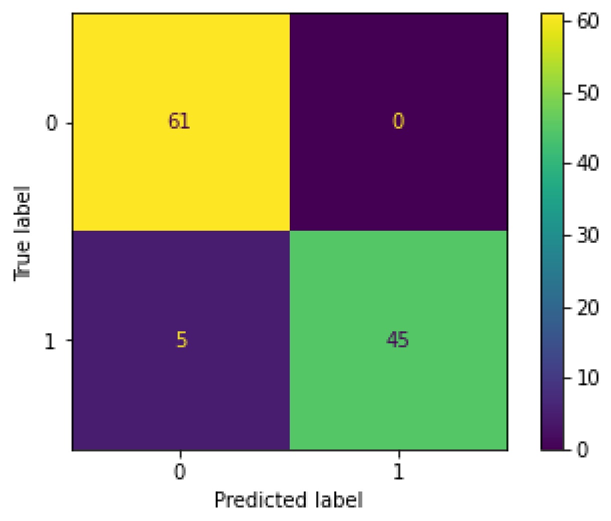
```
<<<<<<<<<< Decision Tree Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.954954954954955
[!] Precision Score: 1.0
[!] Recall Score: 0.9
```

```
In [78]: evaluate_classifier(random_forest,X_test_gm, y_test_gm, "Random Forest")
```

```
<<<<<<<<<< Random Forest Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.954954954954955
[!] Precision Score: 1.0
[!] Recall Score: 0.9
```

All models perform very well on the SP500 Classification task, in particular also a Linear SVM perform well.

This fact involves a linear separability of data.

Mixture Dataset Classification

Experiment setup is similar to 'SP500 Classification' one.

Differences resides in dataset and the use of only kmeans for clustering.

Dataset was built from original Multi-Variate Time Series one, in particular we select the following 5 features:

- SP500
- Dividend
- Earnings
- Consumer Price Index
- Real Price

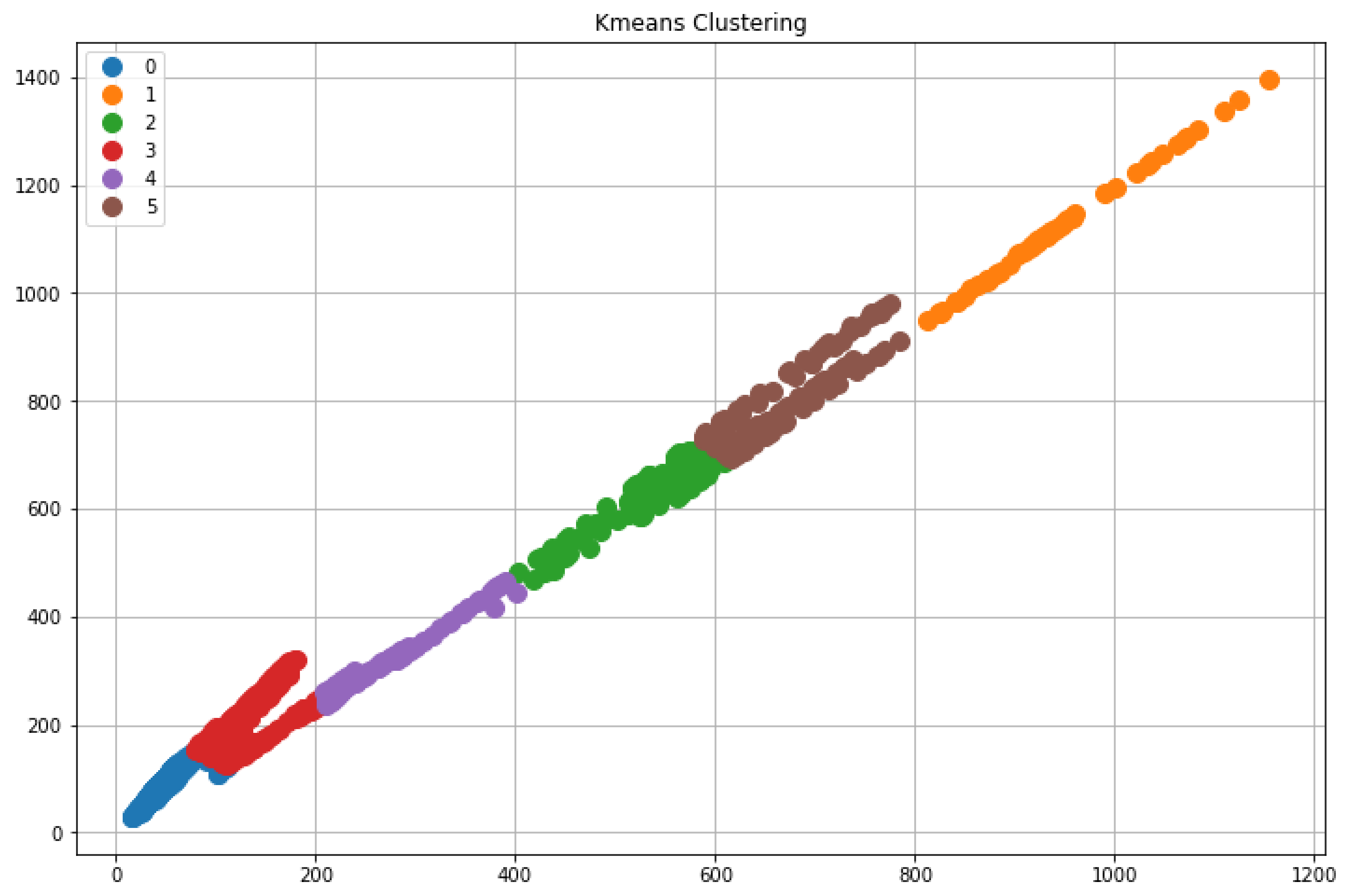
In [79]:
```
#Preparing Data
financial_dataset = full_dataset[['SP500','Dividend' , 'Earnings' , 'Consumer Price Index', 'Real Price']]
financial_dataset = financial_dataset.dropna()
```

Clustering

In [80]:
```python
#Fit Kmeans to data
kmeans = KMeans(n_clusters=6)
kmeans.fit(financial_dataset)


#Evaluate Clusters
kmeans_clusters         = kmeans.labels_.tolist()

labels_0                = [ 1 if label == 0 else 0 for label in kmeans_clusters  ]
labels_1                = [ 1 if label == 1 else 0 for label in kmeans_clusters  ]
labels_2                = [ 1 if label == 2 else 0 for label in kmeans_clusters  ]
labels_3                = [ 1 if label == 3 else 0 for label in kmeans_clusters  ]
labels_4                = [ 1 if label == 4 else 0 for label in kmeans_clusters  ]
labels_5                = [ 1 if label == 5 else 0 for label in kmeans_clusters  ]

ct_labels_0 = reduce( lambda x,y : x+y, labels_0 )
ct_labels_1 = reduce( lambda x,y : x+y, labels_1 )
ct_labels_2 = reduce( lambda x,y : x+y, labels_2 )
ct_labels_3 = reduce( lambda x,y : x+y, labels_3 )
ct_labels_4 = reduce( lambda x,y : x+y, labels_4 )
ct_labels_5 = reduce( lambda x,y : x+y, labels_5 )


print("[!] Kmeans Clustering - N Sample Class 0: {}".format(ct_labels_0))
print("[!] Kmeans Clustering - N Sample Class 1: {}".format(ct_labels_1))
print("[!] Kmeans Clustering - N Sample Class 2: {}".format(ct_labels_2))
print("[!] Kmeans Clustering - N Sample Class 3: {}".format(ct_labels_3))
print("[!] Kmeans Clustering - N Sample Class 4: {}".format(ct_labels_4))
print("[!] Kmeans Clustering - N Sample Class 5: {}".format(ct_labels_5))


#Print Clustering Stats
print("\n\n[!] Kmeans Clustering - Cluster Centers: \n\n{}".format(kmeans.cluster_centers_))


# Plot Clusters
plt.figure(figsize=(12,8))
plt.title("Kmeans Clustering")
plot2d(financial_dataset,kmeans_clusters)
plt.legend()
plt.grid()
plt.show()
```

```
[!] Kmeans Clustering - N Sample Class 0: 1010
[!] Kmeans Clustering - N Sample Class 1: 50
[!] Kmeans Clustering - N Sample Class 2: 90
[!] Kmeans Clustering - N Sample Class 3: 399
[!] Kmeans Clustering - N Sample Class 4: 102
[!] Kmeans Clustering - N Sample Class 5: 113


[!] Kmeans Clustering - Cluster Centers:

[[1.06672574e+01 5.74831683e-01 9.56009901e-01 1.38690693e+01
  1.83032614e+02]
 [2.11948920e+03 4.25024000e+01 9.74872000e+01 2.39453600e+02
  2.20867340e+03]
 [1.02145600e+03 1.95212222e+01 4.43165556e+01 1.92056111e+02
  1.33358167e+03]
 [1.02229599e+02 3.87030075e+00 8.02298246e+00 5.31030075e+01
```

```
  5.01809248e+02]
 [4.35201176e+02 1.24073529e+01 2.32037255e+01 1.40500980e+02
  7.63956667e+02]
 [1.34607752e+03 2.24630973e+01 6.37199115e+01 1.97166018e+02
  1.72061159e+03]]
```



Kmeans clustering return well-separated clusters.

Classification

Data Preparation

In [81]:
```
#Split Train-Test to 75:25
X_train_km, X_test_km, y_train_km, y_test_km = train_test_split(financial_dataset,kmeans_clusters,shuffle=True ,random_state=123)
```

Training

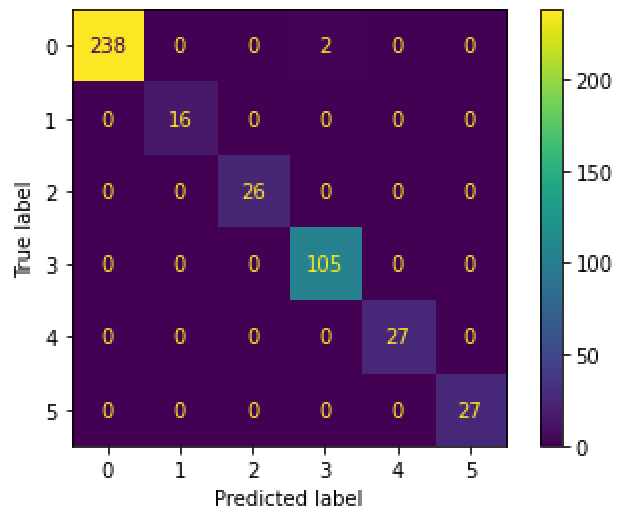
```
In [82]: #Train Models
         linear_svm:SVC                  = SVC(kernel= "linear" , C=1 , max_iter=100)
         linear_svm.fit(X_train_km,y_train_km)

         gaussian_svm:SVC                = SVC(kernel= "rbf" , C=1 , max_iter=100)
         gaussian_svm.fit(X_train_km,y_train_km)


         naive_bayes                     = GaussianNB()
         naive_bayes.fit(X_train_km,y_train_km)


         lg_regressor                    = LogisticRegression( max_iter=100 )
         lg_regressor.fit(X_train_km,y_train_km)


         dec_tree = DecisionTreeClassifier( max_depth=2 )
         dec_tree.fit(X_train_km,y_train_km)


         random_forest                   = RandomForestClassifier(max_depth=2, n_estimators=2)
         random_forest.fit(X_train_km,y_train_km)
```

```
Out[82]: RandomForestClassifier(max_depth=2, n_estimators=2)
```

Evaluation

```
In [83]: evaluate_classifier(linear_svm,X_test_km, y_test_km, "SVM-Linear" , pMultiClass=True)
```

```
<<<<<<<<<< SVM-Linear Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```

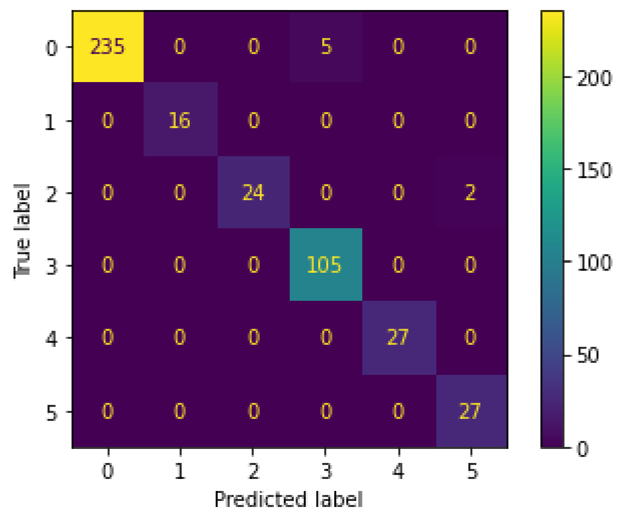


True label \ Predicted label	0	1	2	3	4	5
0	238	0	0	2	0	0
1	0	16	0	0	0	0
2	0	0	26	0	0	0
3	0	0	0	105	0	0
4	0	0	0	0	27	0
5	0	0	0	0	0	27

```
[!] Accuracy Score: 0.9954648526077098
[!] Precision Score: 0.9954648526077098
[!] Recall Score: 0.9954648526077098
```

```
In [84]: evaluate_classifier(gaussian_svm,X_test_km, y_test_km, "SVM-Gaussian", pMultiClass=True)
```

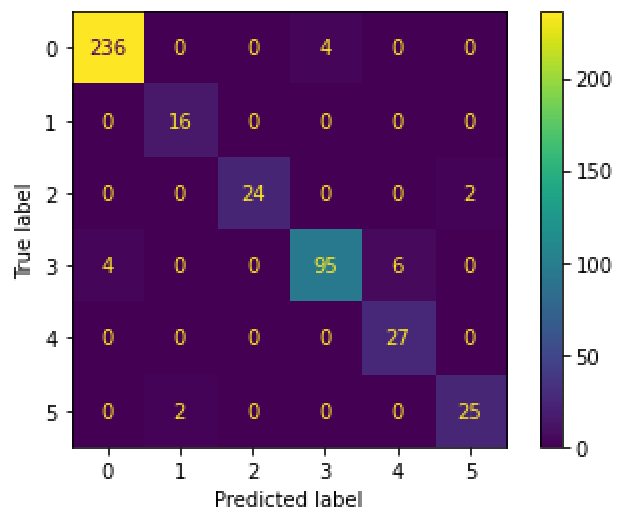
```
<<<<<<<<<< SVM-Gaussian Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.9841269841269841
[!] Precision Score: 0.9841269841269841
[!] Recall Score: 0.9841269841269841
```

```
In [85]: evaluate_classifier(naive_bayes,X_test_km, y_test_km, "Naive Bayes", pMultiClass=True)
```

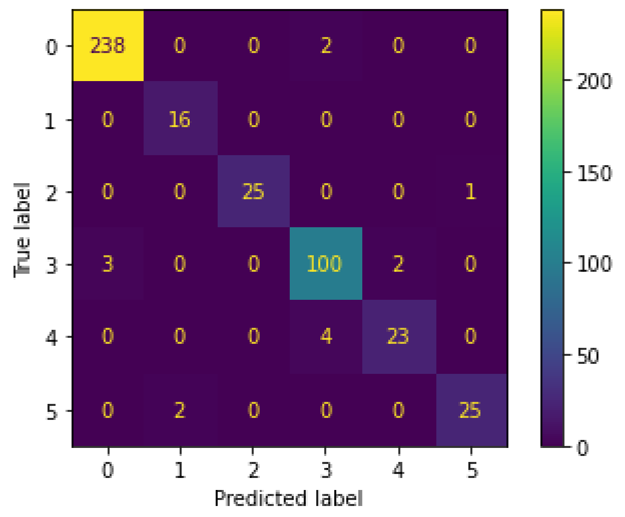
```
<<<<<<<<<< Naive Bayes Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.9591836734693877
[!] Precision Score: 0.9591836734693877
[!] Recall Score: 0.9591836734693877
```

```
In [86]: evaluate_classifier(lg_regressor,X_test_km, y_test_km, "Logistic Regressor", pMultiClass=True)
```

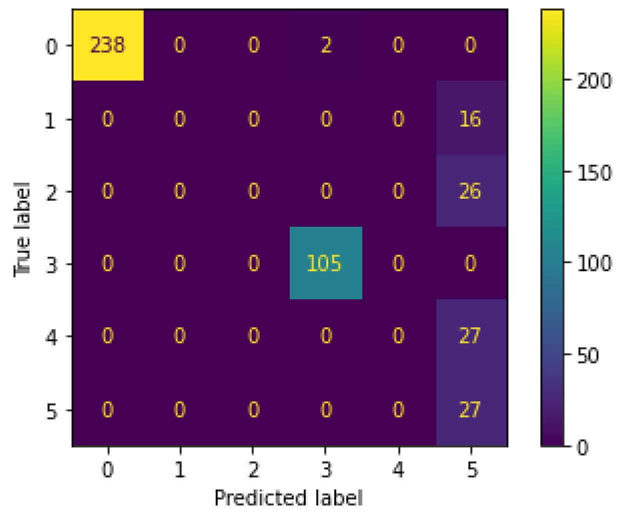
```
<<<<<<<<<<< Logistic Regressor Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.9682539682539683
[!] Precision Score: 0.9682539682539683
[!] Recall Score: 0.9682539682539683
```

```
In [87]: evaluate_classifier(dec_tree,X_test_km, y_test_km, "Decision Tree", pMultiClass=True)
```

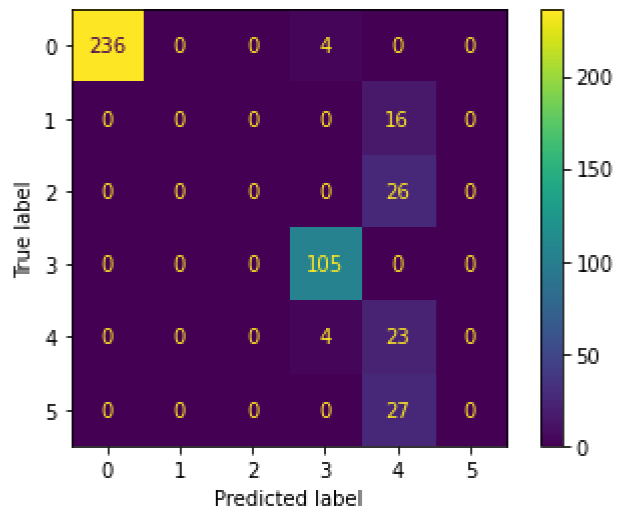
```
<<<<<<<<<<< Decision Tree Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.8390022675736961
[!] Precision Score: 0.8390022675736961
[!] Recall Score: 0.8390022675736961
```

In [88]:
```python
evaluate_classifier(random_forest,X_test_km, y_test_km, "Random Forest", pMultiClass=True)
```

```
<<<<<<<<<< Random Forest Classifier Evaluation >>>>>>>>>>>>

[!] Confusion Matrix:
```



```
[!] Accuracy Score: 0.8253968253968254
[!] Precision Score: 0.8253968253968254
[!] Recall Score: 0.8253968253968254
```

Evaluation results underlines the following facts:

1. SVM (both Linear and Gaussian), Naive-Bayes and Logistic Regressor perform very well on dataset (all up to 95 % of accuracy)
2. Decision Tree and Random Forest perform worse than other models, both are around 83 % of Accuracy.

Conclusion

This project was an important opportunity to study data analysis and in particular Time Series Analysis.

In particular, project outcomes involve the following discoveries:

- SP500 series has a low or null seasonal component
- In recent years of the series (from 1985) there is an important Trend
- SP500 Series is Not Stationary
- SARIMA model is the best forecast model when we consider the Sequential train-test split 80-20 % of our dataset.

  This means that SARIMA best understands overall behaviour of data on long period despite the fact that there isn't real insight into training-data about the Trend that starts from 1985

- Holt-Winters model is the best model in RollingWindow Backtest, this means that model performance are very sensitive to the chengaepoint near year 1985 of our data and more in general sensitive to not seen changepoints.

  In other words, HW fails to predict sudden future trends.

- Prophet model is the best in Expanding Window test, this means that if we give a little insight of "Trend" at the model (case shift from 80 to 90 % of our training-data) it will return right predictions

- In absolute terms (considering all test without distinction), SARIMA is the best model with his **194** of **MAE** error.

  After SARIMA, there is Holt-Winters with **236** of **MAE**.

  At the end, there is Prophet with his **401** of **MAE**.

- Classification Task on SP500 Quarter Data is well-accomplished by Linear Models and not.

  In general, all models do well in this test

- Classification Task on Mixture Dataset underlines outstanding performance for SVM, Logistic Regressor and Bayes.

  Conversely, Trees models of our test trudge with an accuracy of more than 10 % lower than the aforementioned models

References

- [1] (https://www.wiley.com/en-us/Time+Series+Analysis+and+Forecasting+by+Example-p-9780470540640) Bisgaard, Kulahci - Time Series Analysis and Forecasting by Example (First Edition)
- [2] (https://www.springer.com/gp/book/9783319298528) Brockwell, Davis - Introduction to Time Series and Forecasting (Third Edition)
- [3] (https://machinelearningmastery.com/introduction-to-time-series-forecasting-with-python/) Jason Brownlee - Introduction to Time Series Forecasting With Python
- [4] (https://peerj.com/preprints/3190.pdf) Taylor et. al. (Facebook Research) - Forecasting at Scale
- [5] (https://www.researchgate.net/publication/247087596_Out-of_sample_tests_of_forecasting_accuracy_an_analysis_and_review) Tashman - Out-of sample tests of forecasting accuracy: an analysis and review
- [6] (https://robjhyndman.com/papers/ijf25.pdf) Gooijer et. al. - 25 Years of Time Series Forecasting
- [7] (https://en.wikipedia.org/wiki/Local_regression) Local Regression
- [8] (https://www.wessa.net/download/stl.pdf) Cleveland et. al. - STL: A Seasonal-Trend Decomposition Procedure Based on Loess
- [9] (https://datahub.io/core/s-and-p-500#data-cli) Standard and Poor's (S&P) 500 Index Data
- [10] (https://www.scipy.org/) SciPy, python-based ecosystem for mathematics, science, and engineering
- [11] (https://facebookresearch.github.io/Kats/) Kats, one stop shop for time series analysis in Python