Scientific course work 3rd year

Background/Introduction

The Standard & Poor's 500, often abbreviated as the S&P500, is an American stock market index based on the market capitalizations of 500 large companies having common stock listed on the NYSE or NASDAQ.

The S&P 500 index components and their weightings are determined by S&P Dow Jones Indices. It is one of the most commonly followed equity indices, and many consider it one of the best representations of the U.S. stock market, and a bellwether for the U.S. economy.

Why Are Logarithmic Returns Important in Finance?

Logarithmic returns are important in finance because they provide a more accurate measure of the percentage change in the value of an asset over a period of time. This is particularly important when analyzing financial data because the compounding effect of returns over time can have a significant impact on the value of an asset.

The simple return of your portfolio over any time period is the weighted sum of all the simple returns from each of the security. However, we cannot use the same statement with log returns. It will give you an erroneous number. Log return is a continuously compounded rate over time. When we add log returns, we compound. Across different stocks within the same time period, there is no compounding element here.

By using logarithmic returns, we can get a more accurate measure of the percentage change in the value of an asset over a period of time, which is crucial when working with financial data.

What distribution do stock returns follow? We all know that stock market returns are not normally distributed. Instead, we think of them as having fat tails (i.e. extreme events happen more frequently than expected).

```
!pip3 install datetime
!pip3 install yfinance
!pip3 install bs4
!pip3 install os
!pip3 install datetime
!pip3 install pandas
!pip3 install pandas_datareader
!pip3 install statsmodels
!pip3 install scipy
!pip3 install scikit-learn
```

```
Requirement already satisfied: datetime in c:\users\honor\appdata\local\packages\pythonsoftwar
Requirement already satisfied: zope.interface in c:\users\honor\appdata\local\packages\pythons
Requirement already satisfied: pytz in c:\users\honor\appdata\local\packages\pythonsoftwarefou
Requirement already satisfied: setuptools in c:\program files\windowsapps\pythonsoftwarefounda
Requirement already satisfied: yfinance in c:\users\honor\appdata\local\packages\pythonsoftwar
Requirement already satisfied: pandas>=1.3.0 in c:\users\honor\appdata\local\packages\pythonso
Requirement already satisfied: numpy>=1.16.5 in c:\users\honor\appdata\local\packages\pythonso
Requirement already satisfied: requests>=2.31 in c:\users\honor\appdata\local\packages\pythons
```

```
    Requirement already satisfied: multitasking>=0.0.7 in c:\users\honor\appdata\local\packages\py
    Requirement already satisfied: lxml>=4.9.1 in c:\users\honor\appdata\local\packages\pythonsoft
    Requirement already satisfied: appdirs>=1.4.4 in c:\users\honor\appdata\local\packages\pythons
    Requirement already satisfied: pytz>=2022.5 in c:\users\honor\appdata\local\packages\pythonsof
    Requirement already satisfied: frozendict>=2.3.4 in c:\users\honor\appdata\local\packages\pyth
    Requirement already satisfied: peewee>=3.16.2 in c:\users\honor\appdata\local\packages\pythons
    Requirement already satisfied: beautifulsoup4>=4.11.1 in c:\users\honor\appdata\local\packages
    Requirement already satisfied: html5lib>=1.1 in c:\users\honor\appdata\local\packages\pythonso
    Requirement already satisfied: soupsieve>1.2 in c:\users\honor\appdata\local\packages\pythonso
    Requirement already satisfied: six>=1.9 in c:\users\honor\appdata\local\packages\pythonsoftwar
    Requirement already satisfied: webencodings in c:\users\honor\appdata\local\packages\pythonsof
    Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\honor\appdata\local\packages
    Requirement already satisfied: tzdata>=2022.1 in c:\users\honor\appdata\local\packages\pythons
    Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\honor\appdata\local\packag
    Requirement already satisfied: idna<4,>=2.5 in c:\users\honor\appdata\local\packages\pythonsof
    Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\honor\appdata\local\packages\pyt
    Requirement already satisfied: certifi>=2017.4.17 in c:\users\honor\appdata\local\packages\pyt
    Requirement already satisfied: bs4 in c:\users\honor\appdata\local\packages\pythonsoftwarefoun
    Requirement already satisfied: beautifulsoup4 in c:\users\honor\appdata\local\packages\pythons
    Requirement already satisfied: soupsieve>1.2 in c:\users\honor\appdata\local\packages\pythonso
    ERROR: Could not find a version that satisfies the requirement os (from versions: none)
    ERROR: No matching distribution found for os
    Requirement already satisfied: datetime in c:\users\honor\appdata\local\packages\pythonsoftwar
    Requirement already satisfied: zope.interface in c:\users\honor\appdata\local\packages\pythons
    Requirement already satisfied: pytz in c:\users\honor\appdata\local\packages\pythonsoftwarefou
    Requirement already satisfied: setuptools in c:\program files\windowsapps\pythonsoftwarefounda
    Requirement already satisfied: pandas in c:\users\honor\appdata\local\packages\pythonsoftwaref
    Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\honor\appdata\local\packages
    Requirement already satisfied: pytz>=2020.1 in c:\users\honor\appdata\local\packages\pythonsof
    Requirement already satisfied: tzdata>=2022.1 in c:\users\honor\appdata\local\packages\pythons
    Requirement already satisfied: numpy>=1.21.0 in c:\users\honor\appdata\local\packages\pythonso
    Requirement already satisfied: six>=1.5 in c:\users\honor\appdata\local\packages\pythonsoftwar
    Requirement already satisfied: pandas_datareader in c:\users\honor\appdata\local\packages\pyth
    Requirement already satisfied: lxml in c:\users\honor\appdata\local\packages\pythonsoftwarefou
    Requirement already satisfied: pandas>=0.23 in c:\users\honor\appdata\local\packages\pythonsof
    Requirement already satisfied: requests>=2.19.0 in c:\users\honor\appdata\local\packages\pytho
    Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\honor\appdata\local\packages
    Requirement already satisfied: pytz>=2020.1 in c:\users\honor\appdata\local\packages\pythonsof
    Requirement already satisfied: tzdata>=2022.1 in c:\users\honor\appdata\local\packages\pythons
    Requirement already satisfied: numpy>=1.21.0 in c:\users\honor\appdata\local\packages\pythonso
    Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\honor\appdata\local\packag
    Requirement already satisfied: idna<4,>=2.5 in c:\users\honor\appdata\local\packages\pythonsof
    Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\honor\appdata\local\packages\pyt
    Requirement already satisfied: certifi>=2017.4.17 in c:\users\honor\appdata\local\packages\pyt
    Requirement already satisfied: six>=1.5 in c:\users\honor\appdata\local\packages\pythonsoftwar
    Requirement already satisfied: statsmodels in c:\users\honor\appdata\local\packages\pythonsoft
    Requirement already satisfied: numpy>=1.22.3 in c:\users\honor\appdata\local\packages\pythonso
    Requirement already satisfied: scipy!=1.9.2,>=1.8 in c:\users\honor\appdata\local\packages\pyt
```

```
import pandas as pd
import numpy as np
import yfinance as yf
import matplotlib.pyplot as plt
from pandas_datareader import data
import plotly.express as px
import datetime as dt
import seaborn as sns
```

```
sp500 = yf.download('^GSPC', start='1981-05-01', end =dt.datetime.now())
sp500
```

```
[*********************100%%**********************]  1 of 1 completed
```

Date	Open	High	Low	Close	Adj Close	Volume
1981-05-01	0.000000	134.169998	131.429993	132.720001	132.720001	48360000
1981-05-04	0.000000	131.779999	129.610001	130.669998	130.669998	40430000
1981-05-05	0.000000	131.330002	128.929993	130.320007	130.320007	49000000
1981-05-06	0.000000	132.380005	130.089996	130.779999	130.779999	47100000
1981-05-07	0.000000	132.410004	130.210007	131.669998	131.669998	42590000
...	...	...	...	...	...	...
2024-05-03	5122.779785	5139.120117	5101.220215	5127.790039	5127.790039	3924990000
2024-05-06	5142.419922	5181.000000	5142.419922	5180.740234	5180.740234	3683250000
2024-05-07	5187.200195	5200.229980	5178.959961	5187.700195	5187.700195	3987890000
2024-05-08	5168.979980	5191.950195	5165.859863	5187.669922	5187.669922	3842100000
2024-05-09	5189.029785	5215.299805	5180.410156	5214.080078	5214.080078	3727370000

10847 rows × 6 columns

Now, let's extract the historical prices Apple company from yahoo finance using a python library yfinance from 01-05-1981 to the present days/nowadays. (AAPL is one of the 500 S&P 500 stocks)

```
sp500_aapl = yf.download('AAPL', start='1981-05-01', end =dt.datetime.now())
sp500_aapl.head()
```

```
[*********************100%%**********************]  1 of 1 completed
```

Date	Open	High	Low	Close	Adj Close	Volume
1981-05-01	0.126674	0.127790	0.126674	0.126674	0.097898	16553600
1981-05-04	0.126674	0.126674	0.126116	0.126116	0.097467	14448000
1981-05-05	0.126116	0.126116	0.125558	0.125558	0.097036	17539200
1981-05-06	0.122768	0.122768	0.122210	0.122210	0.094448	18950400
1981-05-07	0.123884	0.124442	0.123884	0.123884	0.095742	9363200

```
sp500_aapl.tail()
```

Date	Open	High	Low	Close	Adj Close	Volume
2024-05-03	186.649994	187.000000	182.660004	183.380005	183.380005	163224100
2024-05-06	182.350006	184.199997	180.419998	181.710007	181.710007	78569700
2024-05-07	183.449997	184.899994	181.320007	182.399994	182.399994	77305800
2024-05-08	182.850006	183.070007	181.449997	182.740005	182.740005	45057100
2024-05-09	182.559998	184.660004	182.110001	184.570007	184.570007	48927900

```
# sp500['Volume'].plot(grid=True,figsize=(10,5))
# plt.title('AAPL Volume')
# plt.xlabel('Date')
# plt.ylabel('Volume')
# plt.show()

sp500['Date'] = sp500.index

fig = px.area(
    sp500,
    x='Date',
    y='Volume',
    template='plotly_dark',
    color_discrete_sequence=['cyan'],
    title='S&P index'
)

fig.show()
```

The logarithmic return is a way of calculating the rate of return on an investment. To calculate it you need the initial value of the investment V_i, the final value V_f and the number of time periods t. You then take the natural logarithm of V_f divided by V_i, and divide the result by t:

$$R = \frac{\ln(\frac{V_f}{V_i})}{t} \cdot 100\%$$

This value is normally expressed as a percentage, so you also multiply by 100.

The calculated rate will depend on the value of t that you use. If t is the number of years, then you get an annual rate. This then gives you the continuously compounded annual interest rate that you would need to receive in order to match the return on this investment.

Let p_t denote the price of an asset at time t. Then the return (simple return) of an asset captures these relative movements and is defined as

$$r_t = \frac{p_t - p_{t-1}}{p_{t-1}} = \frac{p_t}{p_{t-1}} - 1$$

In words, a return is the change in price of an asset, relative to its previous value. Note that $p_t > 0$, and therefore $r_t > -1$.

In practice, “returns” often means “log returns”. Log returns are defined as

$$z_t = \log(1 + r_t) = \log(\frac{p_t}{p_{t-1}}) = \log(p_t) - \log(p_{t-1})$$

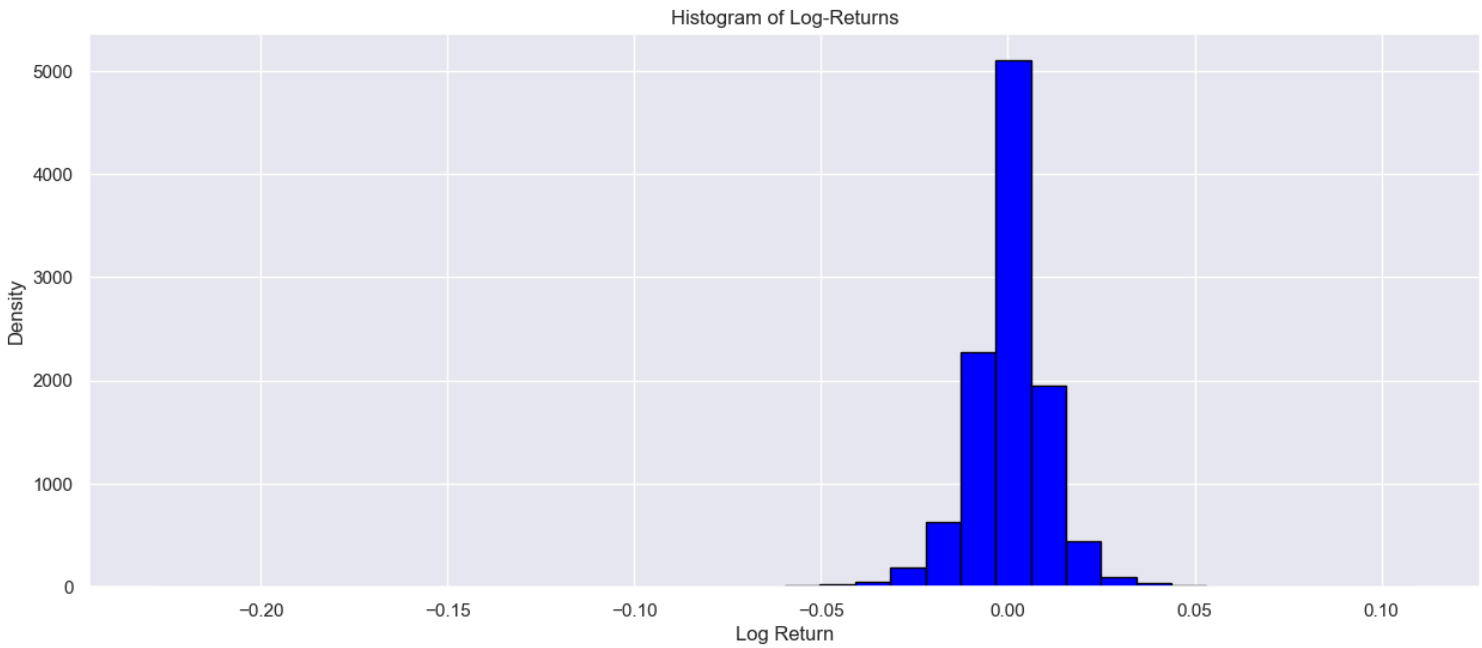
\newline

```
sp500['Log Return'] = np.log(sp500['Adj Close']/sp500['Adj Close'].shift(1))

print(sp500['Log Return'])
```

```
Date
1981-05-01         NaN
1981-05-04   -0.015567
1981-05-05   -0.002682
1981-05-06    0.003523
1981-05-07    0.006782
                 ...
2024-05-03    0.012479
2024-05-06    0.010273
2024-05-07    0.001343
2024-05-08   -0.000006
2024-05-09    0.005078
Name: Log Return, Length: 10847, dtype: float64
```

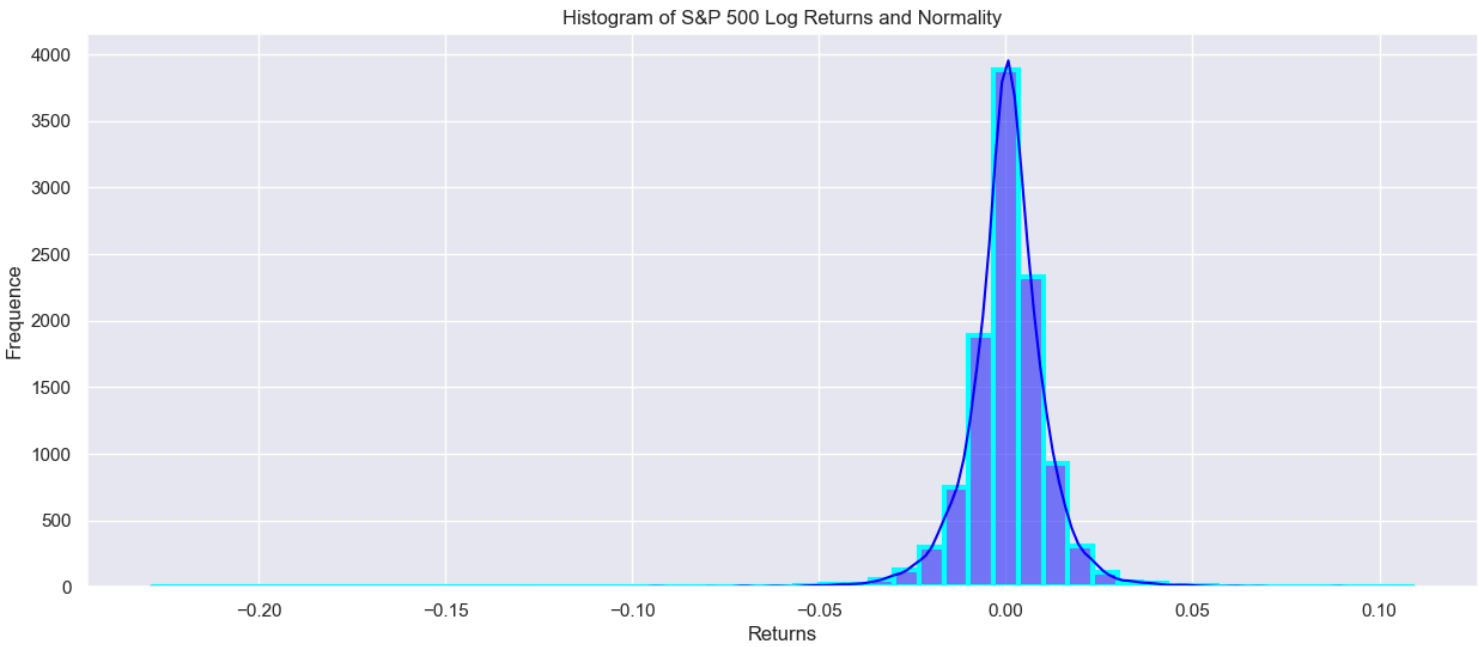
Histogram of log-returns

```
import seaborn as sns

plt.hist(sp500['Log Return'], color = 'blue', edgecolor = 'black',
         bins = int(180/5))

plt.title('Histogram of Log-Returns')
plt.xlabel('Log Return')
plt.ylabel('Density')
plt.show()
```




```
fig = px.histogram(
    sp500,
    x='Log Return',
    color_discrete_sequence=['cyan'],
    title='Histogram of Log-Returns',
    labels={'Log Return': 'Log Return', 'Density': 'Density'},
    template='plotly_dark'
)

fig.show()
```

```
sns.histplot(sp500['Log Return'], bins=int(180/3.5), color='blue',
             edgecolor='cyan', kde=True, linewidth=3)

plt.title('Histogram of S&P 500 Log Returns and Normality')
plt.xlabel('Returns')
plt.ylabel('Frequence')

sns.set_theme(rc={'figure.figsize':(15,6)})

plt.show()
```



In the plot above we see that the tails of the distribution of the S&P 500 log-returns are fatter than the tails of a fitted normal distribution. (so if we increase the amount of bins, we'll see that the histogram of log-returns will be above the plot of the normal distribution)

This means that large moves (both up and down) happens more often than suggested by the model.

The empirical disribution is also commonly peaked around the center, i.e S&P500 historical returns shows non-normal features.

```
# daily_log_returns = sp500['Log Return'].mean()
# daily_log_returns
```

```
# annual_log_returns = sp500['Log Return'].mean() * 250
# print(str(round(annual_log_returns*100, 2)) + ' %')
```

```
# sp500['42d'] = np.round(sp500['Close'].rolling(window=42).mean(),2)
# sp500['252d'] = np.round(sp500['Close'].rolling(window=252).mean(),2)
```

```
## sp500[['Close','42d','252d']].plot(grid=True,figsize=(8,5))
## plt.show()

# sp500['Date'] = sp500.index

# fig = px.line(
#     sp500,
#     x='Date',
#     y=['Close', '42d', '252d'],
#     template='plotly_dark',
#     color_discrete_sequence=['cyan', 'orange', 'green'],
#     title='Close Price with Moving Averages'
# )

# fig.show()


# sp500['Market Returns'] = np.log(sp500['Close'] / sp500['Close'].shift(1))
# sp500['Market Returns'].plot(grid=True,figsize=(8,5))
# plt.show()

sp500['Market Returns'] = np.log(sp500['Close'] / sp500['Close'].shift(1))

fig = px.line(x=sp500.index, y=sp500['Market Returns'], template='plotly_dark', color_discrete_sequen

fig.show()
```

```
sp500.keys()
```

```
Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'Date',
       'Log Return', 'Market Returns'],
      dtype='object')
```

```
sp500['Date'] = sp500.index

fig = px.line(
    sp500,
    x='Date',
    y='Close',
    template='plotly_dark',
    color_discrete_sequence=['cyan', 'orange', 'green'],
    title='Close Price with Moving Averages'
)

fig.show()
```

The first obvious thing to note, aside from the two giant dips at the tail end corresponding to the market crashes in 2020 and 2023, is that the data is clearly non-stationary.(below in this code we'll show it by using some tests)

Below we will try to calculate the first difference of the series, i.e we will subtract the previous value t-1 from the current value t to get the difference.

```
sp500['First Difference'] = sp500['Close'] - sp500['Close'].shift()
# sp500['First Difference'].plot(figsize=(10, 5))
# plt.show()

fig = px.area(
    sp500,
    x='Date',
    y='First Difference',
    template='plotly_dark',
    color_discrete_sequence=['cyan'],
    title='First Difference'
)

fig.show()
```

If we pay attention to the variance, we will see that it's very small early on and steadily increases over time.

This is a sign that the data is not only non-stationary but also exponentially increasing. Now we'll apply a log transform to the original series.

```
sp500['Natural Log'] = sp500['Close'].apply(lambda x: np.log(x))
# sp500['Natural Log'].plot(figsize=(10, 5))
# plt.show()

fig = px.area(
    sp500,
    x='Date',
    y='Natural Log',
    template='plotly_dark',
    color_discrete_sequence=['cyan'],
    title='Natural Log'
)

fig.show()
```

```
sp500['Original Variance'] = sp500['Close'].rolling(window=30, center=True).var()
sp500['Log Variance'] = sp500['Natural Log'].rolling(window=30, center=True).var()

# fig, ax = plt.subplots(2, 1, figsize=(13, 12))
# sp500['Original Variance'].plot(ax=ax[0], title='Original Variance')
# sp500['Log Variance'].plot(ax=ax[1], title='Log Variance')
# fig.tight_layout()

fig = px.area(
    sp500,
    x='Date',
    y='Original Variance',
    template='plotly_dark',
    color_discrete_sequence=['cyan'],
    title='Original Variance'
)

fig.show()

fig = px.area(
    sp500,
    x='Date',
    y='Log Variance',
    template='plotly_dark',
    color_discrete_sequence=['cyan'],
    title='Log Variance'
)

fig.show()
```

Observe that in the top graph, we can't even see any of the variations until the late 80s.

In the bottom graph however it's a different story, changes in the value are clearly visible throughout the entire data set.

From this view, it's clear that our transformation has made the variance relatively constant.

Now we'll calculate the first difference from the logged series, i.e we'll calculate log-returns

```
# Logged First Difference is the column of calculated log-returns
sp500['Logged First Difference'] = sp500['Natural Log'] - sp500['Natural Log'].shift()
sp500['Logged First Difference']
```

```
Date
1981-05-01         NaN
1981-05-04   -0.015567
1981-05-05   -0.002682
1981-05-06    0.003523
1981-05-07    0.006782
                 ...
2024-05-03    0.012479
2024-05-06    0.010273
```

```
2024-05-07    0.001343
2024-05-08   -0.000006
2024-05-09    0.005078
Name: Logged First Difference, Length: 10847, dtype: float64
```

Note: The first value in the table is equal to nan, because $p_t - p_{t-1} = nan$ and the simple efficiency $r_t = \frac{p_t}{p_{t-1}} - 1 = nan$.

We'll use back fill method to replace NaN values, i.e the value of log-returns in first date 1981-05-01

```
# Using back fill method to replace NaN values
sp500['Logged First Difference'] = sp500['Logged First Difference'].fillna(method = 'bfill')
sp500_df = pd.DataFrame(sp500)
sp500_df
```

	Open	High	Low	Close	Adj Close	Volume	Date	Log Return
Date								
1981-05-01	0.000000	134.169998	131.429993	132.720001	132.720001	48360000	1981-05-01	NaN
1981-05-04	0.000000	131.779999	129.610001	130.669998	130.669998	40430000	1981-05-04	-0.015567
1981-05-05	0.000000	131.330002	128.929993	130.320007	130.320007	49000000	1981-05-05	-0.002682
1981-05-06	0.000000	132.380005	130.089996	130.779999	130.779999	47100000	1981-05-06	0.003523
1981-05-07	0.000000	132.410004	130.210007	131.669998	131.669998	42590000	1981-05-07	0.006782
...	...	...	...	...	...	...	...	...
2024-05-03	5122.779785	5139.120117	5101.220215	5127.790039	5127.790039	3924990000	2024-05-03	0.012479
2024-05-06	5142.419922	5181.000000	5142.419922	5180.740234	5180.740234	3683250000	2024-05-06	0.010273
2024-05-07	5187.200195	5200.229980	5178.959961	5187.700195	5187.700195	3987890000	2024-05-07	0.001343
2024-05-08	5168.979980	5191.950195	5165.859863	5187.669922	5187.669922	3842100000	2024-05-08	-0.000006
2024-05-09	5189.029785	5215.299805	5180.410156	5214.080078	5214.080078	3727370000	2024-05-09	0.005078

10847 rows × 14 columns

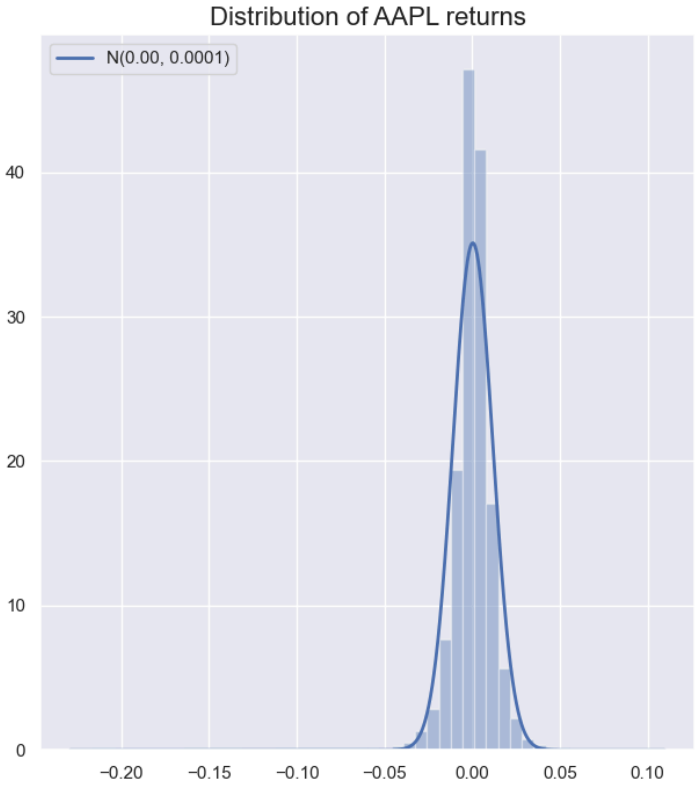
```
# sp500['Logged First Difference'].plot(figsize=(10, 5))
# plt.show()

fig = px.line(sp500, x = 'Date', y = 'Logged First Difference', template='plotly_dark', color_discrete

fig.show()
```

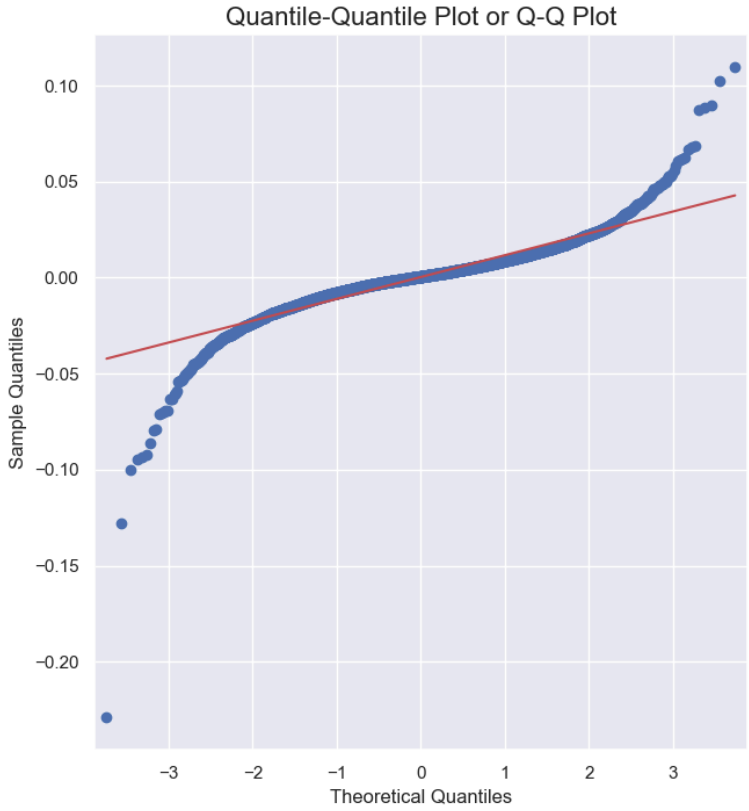
```
import scipy.stats as scs

# Obtaining range of the plot
plot_range = np.linspace(sp500['Logged First Difference'].min(), sp500['Logged First Difference'].max(

# Obtaining the mean
mu = sp500['Logged First Difference'].mean()

# Obtaining the standard deviation
sigma = sp500['Logged First Difference'].std()

# Obtaining the probability distribution function of the log returns series
pdf_series = scs.norm.pdf(plot_range, loc=mu, scale=sigma)

print((mu, sigma))
```

```
(0.0003369880786848332, 0.011369022336406104)
```

Test for normality

```
import statsmodels.api as sm
import scipy.stats as scs

fig, ax = plt.subplots(1, 2, figsize=(16, 8))

# kde : Whether to plot a gaussian kernel density estimate. Setting it to False to disable that
# norm_hist : produces density curve instead of count
sns.distplot(sp500['Logged First Difference'].values, kde=False, norm_hist=True, ax=ax[0])

ax[0].set_title('Distribution of AAPL returns', fontsize=16)
ax[0].plot(plot_range, pdf_series, 'b', lw=2, label=f'N({mu:.2f}, {sigma**2:.4f})')
ax[0].legend(loc='upper left');

qq = sm.qqplot(sp500['Logged First Difference'].values, line='s', ax=ax[1])
ax[1].set_title('Quantile-Quantile Plot or Q-Q Plot', fontsize = 16)
plt.show()
```






```
import plotly.graph_objs as go
import numpy as np
import scipy.stats as stats

# Quantile's Calculation
qq = stats.probplot(sp500['Logged First Difference'], dist="norm")
x = np.array([qq[0][0][0], qq[0][0][-1]])

# Q-Q plot
fig = go.Figure()
fig.add_trace(go.Scatter(x=qq[0][0], y=qq[0][1], mode='markers', name='Empirical Data'))
fig.add_trace(go.Scatter(x=x, y=qq[1][1] + qq[1][0]*x, mode='lines', name='Theoretical Line'))
fig.update_layout(title='Q-Q Plot of Logged First Difference', xaxis_title='Theoretical Quantiles', ya
fig.show()
```

From Q-Q Plot we can see that the tails of the distribution of the S&P 500 log-returns are fatter than the tails of a fitted normal distribution, i.e log returns have fatter tails than the normal distribution; while log returns are

slightly more symmetric. And we can also see asymmetricy from the plot, the left tail is heavier.

This means that large moves (both up and down) happens more often than suggested by the model.

The empirical disribution is also commonly peaked around the center, i.e S&P 500 historical returns shows non-normal features.

In conclusion, we showed that S&P 500 log-returns are not normally distributed.

```
fig = px.box(sp500, y='Logged First Difference', width=500, height=600)
fig.update_layout(title='Box Plot of Logged First Difference')
fig.show()
```

Box Plot also know as a box and whisker plot is another way to visualize the normality of a variable. It displays the distribution of data based on a five-number summary i.e. minimum, first quartile (Q1), median (Q2), third quartile (Q3) and maximum.

Desription from seaborn.boxplot (A box plot (or box-and-whisker plot) shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution, except for points that are determined to be “outliers” using a method that is a function of the inter-quartile range.)

```
# the Kolmogorov-Smirnov test
ks_statistic, ks_p_value = stats.kstest(sp500['Logged First Difference'], 'norm', args = (sp500['Logg
print(f'Kolmogorov-Smirnov statistic: {ks_statistic}')
print(f'Kolmogorov-Smirnov p-value: {ks_p_value}')
alpha = 0.05
if ks_p_value < alpha:
    print('The null hypothesis can be rejected. Probably not Gaussian')
else:
    print('The null hypothesis cannot be rejected. Probably Gaussian')
```

```
Kolmogorov-Smirnov statistic: 0.08541568981545039
Kolmogorov-Smirnov p-value: 2.6754788800311997e-69
The null hypothesis can be rejected. Probably not Gaussian
```

The Kolmogorov Smirnov test computes the distances between the empirical distribution and the theoretical distribution and defines the test statistic as the supremum of the set of those distances.

The Test Statistic of the KS Test is the Kolmogorov Smirnov Statistic, which follows a Kolmogorov distribution if the null hypothesis is true. If the observed data perfectly follow a normal distribution, the value of the KS statistic will be 0. The P-Value is used to decide whether the difference is large enough to reject the null hypothesis:

The advantage of this is that the same approach can be used for comparing any distribution, not necessary the normal distribution only.

\newline

Calculating skewness and kurtosis of the log returns series

```
min = sp500['Logged First Difference'].min()
max = sp500['Logged First Difference'].max()
median = sp500['Logged First Difference'].median()
mean = sp500['Logged First Difference'].mean()

print("Minimum of the log return series is", round(min, 3))
print("Maximum of the log return series is", round(max, 3))
print("Mean of the log return series is", round(median, 5))
print("Median of the log return series is", round(mean, 5))
```

```
Minimum of the log return series is -0.229
Maximum of the log return series is 0.11
Mean of the log return series is 0.00055
Median of the log return series is 0.00034
```

```
s = sp500['Logged First Difference'].skew()
print("Skewness of the log return series is", round(s, 2))
```

```
Skewness of the log return series is -1.11
```

Skewness is the third standardized moment and it measures the lack of symmetry in a distribution.

The skewness of the Gaussian distribution is zero and any other symmetrical distribution should have a skewness of close to zero also.

Negative values of skewness indicate that the data is skewed left, or negatively skewed while positive values tell the data being skewed right, or positively skewed.

```
k = sp500['Logged First Difference'].kurtosis()
print("Kurtosis of the log return series is", round(k, 2))
```

```
Kurtosis of the log return series is 24.81
```

The kurtosis is the fourth standardized moment of the distribution and for the normal distribution it is exactly three.

Therefore for reasonably well behaved distributions a kurtosis figure higher than 3 it indicates that the distribution has heavy tails and peaks close to its mean.

If less than 3 the kurtosis tells us that the sample data has a flatter distribution than the normal.

\newline

Above we can see that the kurtosis is 24.81 which is larger than normal distribution, which kurtosis = 3.

So the S&P 500 return has a heavier tail than normal. The skewness is -1.11, which means the distribution of return is asymmetric and the negative value implies that the distribution has a long left tail.

\newline

Чтобы изменить содержимое ячейки, дважды нажмите на нее (или выберите "Ввод")

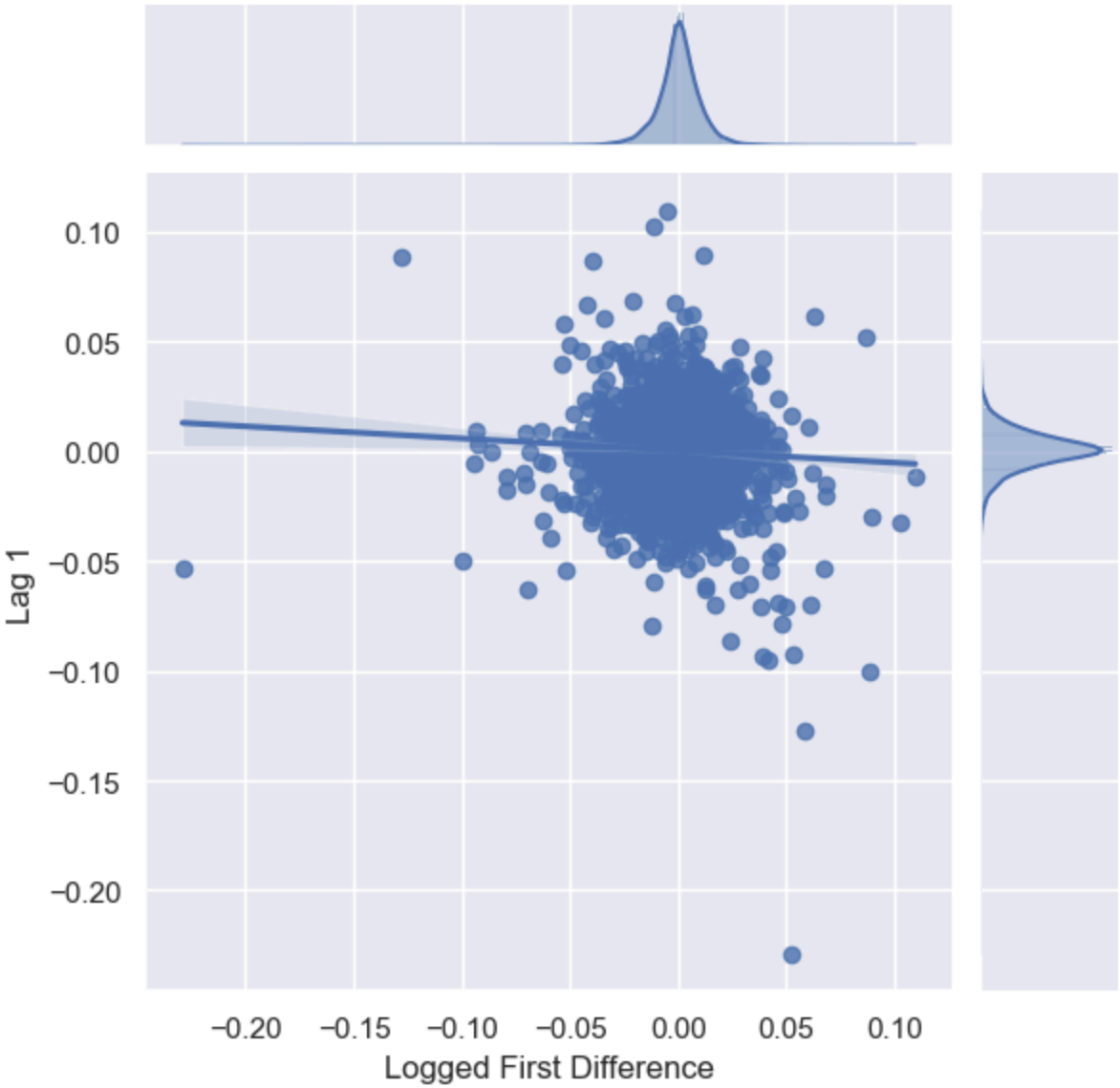
Obviously, these data have a greater variance before 2010 than after 2010.

Below we'll create some lag variables y(t-1), y(t-2) etc. and examine their relationship to y(t) and look at 1 and 2-day lags along with weekly and monthly lags to look for "seasonal" effects.

```
sp500['Lag 1'] = sp500['Logged First Difference'].shift()
sp500['Lag 2'] = sp500['Logged First Difference'].shift(2)
sp500['Lag 5'] = sp500['Logged First Difference'].shift(5)
sp500['Lag 30'] = sp500['Logged First Difference'].shift(30)
```

One interesting visual way to evaluate the relationship between lagged variables is to do a scatter plot of the original variable vs. the lagged variable and see where the distribution lies with a joint plot using the seaborn package.

```
sns.jointplot(data=sp500, x='Logged First Difference', y='Lag 1', palette='Set2', kind = 'reg')
plt.show()
```



Notice how tightly packed the mass is around 0, besides it also appears to be pretty evenly distributed - the marginal distributions on both axes are roughly normal.

This seems to indicate that knowing the index value one day doesn't tell us much about what it will do the next day.

There's very little correlation between the change in value from one day to the next.

```
from statsmodels.tsa.stattools import acf
from statsmodels.tsa.stattools import pacf

lag_correlations = acf(sp500['Logged First Difference'].iloc[1:])
lag_partial_correlations = pacf(sp500['Logged First Difference'].iloc[1:])
```

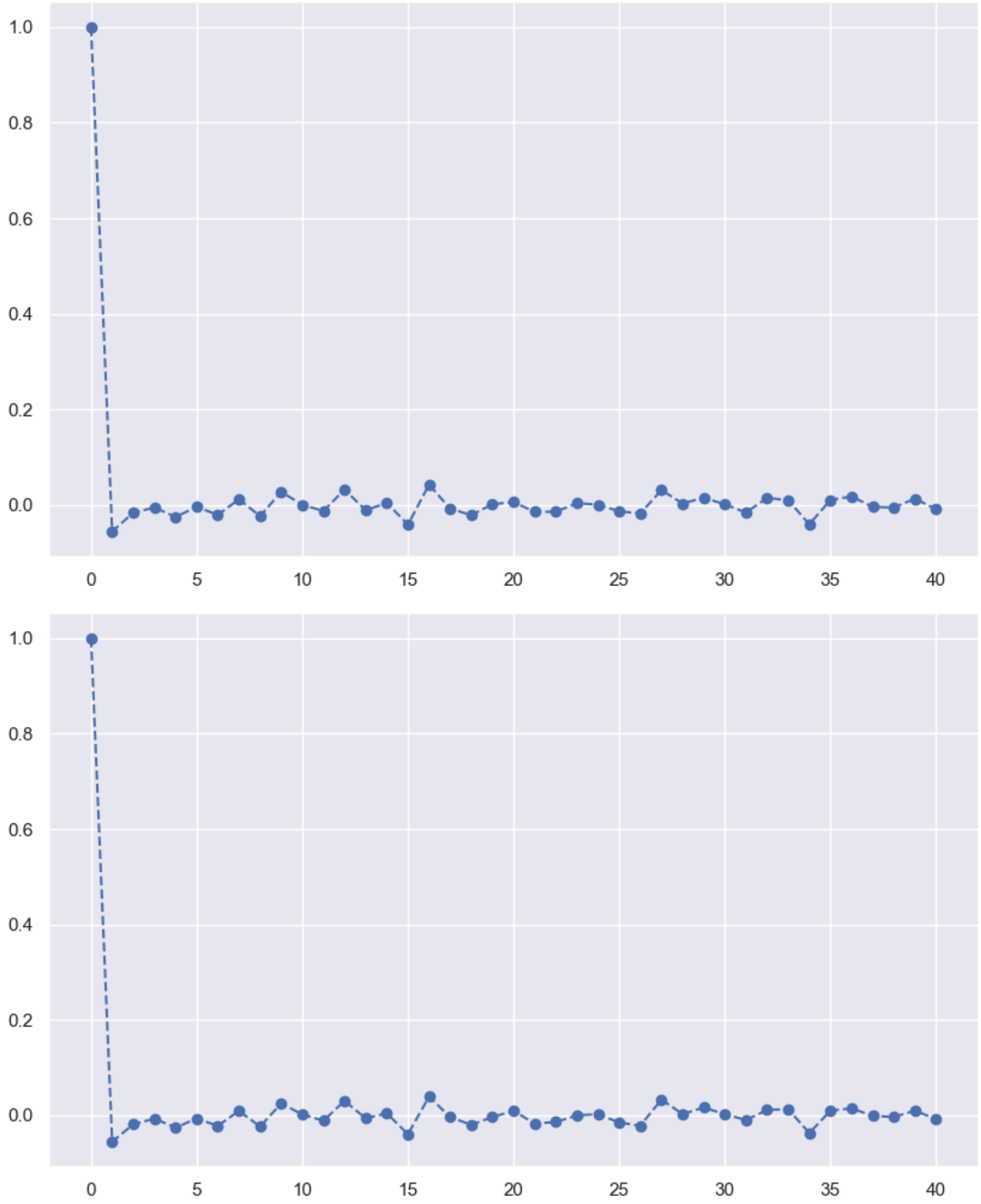
The auto-correlation function computes the correlation between a variable and itself at each lag step up to some limit (in this case 40).

The partial auto-correlation function computes the correlation at each lag step that is NOT already explained by previous, lower-order lag steps.

We can plot the results to see if there are any significant correlations.

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(lag_correlations, marker='o', linestyle='--')

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(lag_partial_correlations, marker='o', linestyle='--')
```

[<matplotlib.lines.Line2D at 0x1d68c830490>]



```
correlations_df = pd.DataFrame({
    'Lag': range(len(lag_correlations)),
    'ACF': lag_correlations,
    'PACF': lag_partial_correlations
})

# ACF plot
fig_acf = px.area(correlations_df, x='Lag', y='ACF', title='Autocorrelation Function (ACF)')
fig_acf.show()

# PACF plot
fig_pacf = px.area(correlations_df, x='Lag', y='PACF', title='Partial Autocorrelation Function (PACF)
fig_pacf.show()
```

The autocorrelation and partial-autocorrelation results are very close to each other (the first upper plot is the autocorrelation results and the second lower plot is the partial-autocorrelation results).

What this shows is that there is no significant (> 0.2) correlation between the value at time t and at any time prior to t up to 40 steps behind.

In order words, the log-returns series is just a sequence close to independent and identically distributed (i.i.d).

\newline \newline

```
# the Jarque-Bera Test

import statsmodels.api as sm

jb_stat, jb_p_value, skew, kurtosis = sm.stats.stattools.jarque_bera(sp500['Logged First Difference']
print(f'Jarque-Bera statistic: {jb_stat}')
print(f'Jarque-Bera p-value: {jb_p_value}')
print(f'Skewness: {skew}')
print(f'Kurtosis: {kurtosis}')
alpha = 0.05
if jb_p_value < alpha:
    print('The null hypothesis can be rejected. Probably not Gaussian')
else:
    print('The null hypothesis cannot be rejected. Probably Gaussian')
```



```
Jarque-Bera statistic: 280220.20958531136
Jarque-Bera p-value: 0.0
Skewness: -1.110356492164379
Kurtosis: 27.80084281557391
The null hypothesis can be rejected. Probably not Gaussian
```

The Jarque-Bera test examines the skewness and kurtosis of the data sample to see if it matches that of the normal distribution.

It is one of the simplest and very likely the most commonly used procedure for testing normality of financial time series returns.

It offers a joint test of the null hypothesis of normality in that the sample skewness equals zero and the sample kurtosis equals three.

```
from scipy.stats import shapiro

sw_stat, sw_p_value = stats.shapiro(sp500['Logged First Difference'])
print(f'Jarque-Bera statistic: {jb_stat}')
print(f'Jarque-Bera p-value: {jb_p_value}')
#print('stat=%.3f, p=%.3f' % (sw_stat, p))
alpha = 0.05
if sw_p_value < alpha:
    print('The null hypothesis can be rejected. Probably not Gaussian')
else:
    print('The null hypothesis cannot be rejected. Probably Gaussian')
```

```
Jarque-Bera statistic: 280220.20958531136
Jarque-Bera p-value: 0.0
The null hypothesis can be rejected. Probably not Gaussian
```

The Shapiro-Wilk test is a test for the rank correlation between the empirical data and that of a sample from a normal distribution.

It is a regression test that compares an estimate of the empirical standard deviation using a linear combination of the order statistics to the theoretical normal estimate.

It concentrates on the slope of a plot of the order statistics versus the expected normal order statistics.

\newline

We can clearly state that the Log Returns series of S&P 500 stock does not follow Gaussian distribution.

Also, negative skewness, high positive kurtosis value and high Jarque-Bera test (or Shapiro-Wilk test, or Kolmogorov-Smirnov test) statistic proves that.

```
# the Augmented Dickey-Fuller (ADF) test

from statsmodels.tsa.stattools import adfuller

result = adfuller(sp500['Logged First Difference'])
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))
if result[0] < result[4]["5%"]:
    print ("Reject H_0 - Time Series is Stationary")
else:
    print ("Failed to Reject H_0 - Time Series is Non-Stationary")
```

```
ADF Statistic: -18.532929
p-value: 0.000000
Critical Values:
        1%: -3.431
        5%: -2.862
        10%: -2.567
Reject H_0 - Time Series is Stationary
```

The Augmented Dickey-Fuller Test is a hypothesis test.

The null-hypothesis is that the time series is non-stationary, and the alternative is that the series is stationary.

Thus, we need to find a p-value low enough to reject our null hypothesis, thus suggesting the series is stationary.

In conclusion, we see that Log-returns of the S&P 500 is stationary.

\newline

Extrapolation (application of the block quantiles method and its comparison with GPD+MLE)

\newline

```
#Here we import some necessary libraries. Please, run this cell.
import warnings
import scipy.stats as ss
import scipy.optimize as opt
import statsmodels.formula.api as smf
import statistics
import itertools
import math
import random
warnings.filterwarnings("ignore")

import statsmodels.formula.api as smf
import statsmodels.stats.api as sms
from statsmodels.graphics.regressionplots import plot_leverage_resid2
from scipy.optimize import OptimizeResult
from sklearn.linear_model import QuantileRegressor
from tqdm import tqdm
```

```
#Here are all the functions that I use below

def quantile_LW_Pareto(n, k, glob):
  q = np.log(np.log(n/k))
  r = np.sqrt(glob**2 + 4*q)
  return np.exp(np.exp(((r - glob)/2)**2))

def quantile_W_LW(n, k, glob):
  return np.exp((np.log(np.log(n/k)))**2/glob)

def quantile_RV_W(n, k, glob):
  return np.exp(np.sqrt(np.log(n/k)/glob))

def test_statistic(n, k_n, samples, glob, method):

    const_log = np.log(n*1./k_n)

    if method == "LW_Pareto":
      #usually glob = 1.1 in this case
      const_quant = quantile_LW_Pareto(n, k_n, glob)
      const_quant2 = quantile_LW_Pareto(n, 2*k_n, glob)

      def locscale_LW_Pareto(x):
        t = (x - samples[n-k_n])/(samples[n-k_n] - samples[n-2*k_n])*(const_quant - const_quant2) +
        return np.exp(glob*np.sqrt(np.log(np.log(t))))*np.log(t) - const_log

      tmp = np.sum(locscale_LW_Pareto(np.array(samples[n-k_n:n])))

    elif method == "W_LW":
      #usually glob \in [3; 4] in this case
      const_quant = quantile_W_LW(n, k_n, glob)
      const_quant2 = quantile_W_LW(n, 2*k_n, glob)

      def locscale_W_LW(x):
        t = (x - samples[n-k_n])/(samples[n-k_n] - samples[n-2*k_n])*(const_quant - const_quant2) +
        return np.exp(np.sqrt(glob*abs(np.log(t)))) - const_log

      tmp = np.sum(locscale_W_LW(np.array(samples[n-k_n:n])))

    elif method == "RV_W":
      #usual value for glob is 0.4
      const_quant = quantile_RV_W(n, k_n, glob)
      const_quant2 = quantile_RV_W(n, 2*k_n, glob)

      def locscale_RV_W(x):
        t = (x - samples[n-k_n])/(samples[n-k_n] - samples[n-2*k_n])*(const_quant - const_quant2) +
        return glob*np.power(np.log(t),2) - const_log

      tmp = np.sum(locscale_RV_W(np.array(samples[n-k_n:n])))

    return 1/np.sqrt(k_n)*(tmp - k_n)

#Some necessary functions to apply block quantiles method and classical methods for tail estimation
def block_maxima(sample, number_of_blocks):
```

```
    n = len(sample)
    length = n // number_of_blocks
    #length = number_of_blocks
    sample_new = list(map(lambda x: np.max(sample[int(length*x):int(length*(x+1))]), range(number_of

    return sample_new
################################################################################
def generate_model(sample, alpha_min, alpha_max, block_min, block_max, number_block, ratio, threshol

    if is_threshold:
      sample = sample[sample>threshold] - threshold
    n = len(sample)

    #block_size = list(map(lambda x: block_min + int(x*(block_max - block_min)/(number_block-1)), ra
    block_size = list(map(lambda x: int(block_min*np.power((block_max - block_min), x/(number_block-
    #alphas = list(map(lambda x: alpha_min + x*(alpha_max - alpha_min)/(number_alpha-1), range(numbe

    feature = []
    target = []
    index = 0

    for i in block_size:
        n_temp = n // i
        alphas = []
        sample_temp = block_maxima(sample, n_temp)
        #sample_temp = block_maxima(sample, i)
        sample_temp = sorted(sample_temp)

        for j in range(n_temp):
          if (j >= alpha_min*n_temp)&(j <= alpha_max*n_temp):
            alphas.append(j)

        alphas = list(alphas)
        #feature_temp = list(map(lambda x: 1 - np.power(x, 1/n_temp), alphas))
        feature[index:index+len(alphas)] = list(map(lambda x: (1 - np.power(x/n_temp, 1/i)), alphas)
        #target[j*number_alpha:(j+1)*number_alpha] = list(map(lambda x: sample_temp[int(x*n_temp-1)]
        target[index:index+len(alphas)] = list(map(lambda x: sample_temp[x-1], alphas))
        #list(map(lambda x: sample_temp[int(x*n_temp-1)]*((x*n_temp) % 1) + sample_temp[int(x*n_temp
        index = index+len(alphas)

    return feature, target
################################################################################
def quantile_of_maxima(evi, sigma, ratio, threshold, level, n):
    return threshold + np.exp(sigma)*np.power((1 - np.power(level, 1/n)), evi)
################################################################################
def quantile_classical(evi, sigma, theta, ratio, threshold, level, n):
    if evi != 0:
      return threshold + (np.power((1 - np.power(level, 1/(n*theta)))/ratio, -evi) - 1)*sigma/evi
    else:
      return threshold - np.log((1 - np.power(level, 1/(n*theta)))/ratio)*sigma
################################################################################
def quantile_trivial(evi, sigma, level, n):
    return np.exp(sigma + evi*np.log(1 - np.power(level, 1/n)))
################################################################################
def quantile_logweibull(theta, scale, location, level, n):
```

```
    return location + scale*np.exp(np.power(-np.log(1 - np.power(level, 1/n)), theta))
###############################################################################
def quantile_logweibull_trivial(theta, sigma, level, n):
    return np.exp(sigma*np.power(-np.log(1 - np.power(level, 1/n)), theta))
###############################################################################
def quantile_weibull(theta, scale, location, level, n):
    return location + scale*np.power(-np.log(1 - np.power(level, 1/n)), theta)
###############################################################################
def quantile_weibull_trivial(theta, sigma, level, n):
    return sigma*np.power(-np.log(1 - np.power(level, 1/n)), theta)
###############################################################################
def quantile_gev(evi, scale, location, theta, level, n):
    return location + scale*(-1 + np.power(-np.log(level)/n/theta, -evi))/evi
###############################################################################
def hill_estimator(sample, ratio):
    sample = sorted(sample)
    n = len(sample)
    return np.mean(np.log(sample[int(n*(1-ratio)):])) - np.log(sample[int(n*(1-ratio))-1])
###############################################################################
def moment(sample, ratio, power):
    sample = sorted(sample)
    n = len(sample)
    threshold = sample[int(n*(1-ratio))-1]
    return np.mean(np.power(np.log(sample[int(n*(1-ratio)):]) - np.log(threshold), power))
###############################################################################
def moment_estimator(sample, ratio):
    hill = hill_estimator(sample, ratio)
    second_moment = moment(sample, ratio, 2)
    return hill + 1 - 0.5*second_moment/(second_moment - np.power(hill, 2))
###############################################################################
def mle_estimator(sample, ratio):
    sample = sorted(sample)
    n = len(sample)
    threshold = sample[int(n*(1 - ratio))]
    return ss.genpareto.fit(np.array(sample[int(n*(1 - ratio))+1:]) - threshold, floc = 0)
###############################################################################
def sigma_estimator(sample, ratio):

    sample = sorted(sample)
    n = len(sample)
    gamma = hill_estimator(sample, ratio)
    high_level = sample[int(n*(1-ratio))-1]

    ran = range(int(n*ratio))
    m_2n = 0
    for i in ran:
       m_2n += (np.log(sample[int(n-i-1)]) - np.log(high_level))**2/int(n*ratio)
    #m_2n = np.mean(list(map(lambda x: np.power(np.log(sample[int(x)]/high_level), 2), np.arange(int
    gamma_minus = 1 - 0.5*m_2n/(m_2n - gamma**2)

    return high_level*gamma*(1-gamma_minus)
###############################################################################3333
def adam(
    fun,
    x0,
```

```
    jac,
    args=(),
    learning_rate=0.005,
    beta1=0.9,
    beta2=0.999,
    eps=1e-8,
    startiter=0,
    maxiter=1000,
    callback=None,
    **kwargs
):
    """``scipy.optimize.minimize`` compatible implementation of ADAM -
    [http://arxiv.org/pdf/1412.6980.pdf].
    Adapted from ``autograd/misc/optimizers.py``.
    """
    x = x0
    m = np.zeros_like(x)
    v = np.zeros_like(x)

    for i in range(startiter, startiter + maxiter):
        g = jac(x)

        if callback and callback(x):
            break

        m = (1 - beta1) * g + beta1 * m  # first  moment estimate.
        v = (1 - beta2) * (g**2) + beta2 * v  # second moment estimate.
        mhat = m / (1 - beta1**(i + 1))  # bias correction.
        vhat = v / (1 - beta2**(i + 1))
        x = x - learning_rate * mhat / (np.sqrt(vhat) + eps)

    i += 1
    return OptimizeResult(x=x, fun=fun(x), jac=g, nit=i, nfev=i, success=True)
################################################################################

def CvM(x,y):
  res = ss.cramervonmises_2samp(x, y)
  return res.statistic#, res.pvalue
################################################################################
def LS(feature, target, method, N, alphas):

    target = np.array(target)
    feature = np.array(feature)
    mintarget = target.min() - math.e - 1
    target -= mintarget
    d1 = {'target': target, 'feature': feature}
    data2 = pd.DataFrame(d1)

    if method == 'W':
        m = smf.ols('np.log(target) ~ np.log(-np.log(feature))', data=data2)
        mfit = m.fit()
        quantiles = np.array(list(map(lambda x: quantile_weibull_trivial(mfit.params[1], np.exp(mfit
    elif method == 'LW':
        m = smf.ols('np.log(np.log(target)) ~ np.log(-np.log(feature))', data=data2)
        mfit = m.fit()
```

```
        quantiles = np.array(list(map(lambda x: quantile_logweibull_trivial(mfit.params[1], np.exp(m
    elif method == 'RV':
        m = smf.ols('np.log(target) ~ np.log(feature)', data=data2)
        mfit = m.fit()
        quantiles = np.array(list(map(lambda x: quantile_trivial(mfit.params[1], mfit.params[0], x,

    #return quantiles, mfit.params[1]
    return quantiles, mfit.params[0], mfit.params[1]
################################################################################
def Theil(feature, target, method, N, alphas):

    target = np.array(target)
    feature = np.array(feature)
    mintarget = target.min() - math.e - 1.0
    target -= mintarget

    if method == 'W':
        x = np.log(-np.log(feature))
        y = np.log(target)
    elif method == 'LW':
        x = np.log(-np.log(feature))
        y = np.log(np.log(target))
    elif method == 'RV':
        x = np.log(feature)
        y = np.log(target)

    #diff = []
    #for k in range(len(x)):
    #  for j in range(k+1, len(x)):
    #    diff.append((y[k] - y[j])/(x[k] - x[j]))

    evi_theil, sigma_theil, a, b = ss.theilslopes(y, x, alpha=0.95, method='joint')

    #evi_theil = statistics.median(diff)
    #sigma_theil = statistics.median(y - evi_theil*x)

    if method == 'W':
        quantiles = np.array(list(map(lambda x: quantile_weibull_trivial(evi_theil, np.exp(sigma_the
    elif method == 'LW':
        quantiles = np.array(list(map(lambda x: quantile_logweibull_trivial(evi_theil, np.exp(sigma_
    elif method == 'RV':
        quantiles = np.array(list(map(lambda x: quantile_trivial(evi_theil, sigma_theil, x, N), alph

    #return quantiles, evi_theil
    return quantiles, sigma_theil, evi_theil
################################################################################
def Adam(feature, target, method, N, alphas, starting_theta, rho):

    target = np.array(target)
    #target_std = 1
    #target_mean = 0
    #For tail estimation it is important!
    target_mean = target.mean()
    target_std = target.std()
    target = (target - target_mean)/target_std
```

```
    target_min = target.min()-1.

    if method == 'W':
      def func(x):
        return np.sum(np.power(np.abs(target - x[2] - x[1]*np.power(-np.log(feature), x[0])), rho))/

      def gradient(x):
        d = np.zeros(3)
        d[0] = - np.sum(np.sign(target - x[2] - x[1]*np.power(-np.log(feature), x[0]))*x[1]*np.power
        d[1] = - np.sum(np.sign(target - x[2] - x[1]*np.power(-np.log(feature), x[0]))*np.power(-np.
        d[2] = - np.sum(np.sign(target - x[2] - x[1]*np.power(-np.log(feature), x[0]))*np.power(np.a
        return d

    elif method == 'LW':
      def func(x):
        return np.sum(np.power(np.abs(target - x[2] - x[1]*np.exp(np.power(-np.log(feature), x[0]))

      def gradient(x):
        d = np.zeros(3)
        d[0] = - np.sum(np.sign(target - x[2] - x[1]*np.exp(np.power(-np.log(feature), x[0])))*x[1]*
        d[1] = - np.sum(np.sign(target - x[2] - x[1]*np.exp(np.power(-np.log(feature), x[0])))*np.ex
        d[2] = - np.sum(np.sign(target - x[2] - x[1]*np.exp(np.power(-np.log(feature), x[0])))*np.po
        return d

    elif method == 'RV':
      def func(x):
        return np.sum(np.power(np.abs(target - x[2] - x[1]*np.power(feature, - x[0])), rho))/rho

      def gradient(x):
        d = np.zeros(3)
        d[0] = np.sum(np.sign(target - x[2] - x[1]*np.power(feature, - x[0]))*x[1]*np.power(feature,
        d[1] = - np.sum(np.sign(target - x[2] - x[1]*np.power(feature, - x[0]))*np.power(feature, -
        d[2] = - np.sum(np.sign(target - x[2] - x[1]*np.power(feature, - x[0]))*np.power(np.abs(targ
        return d

    #z = np.array([starting_theta, 1., 0])
    z = np.array([starting_theta, 1., -1])
    #z = np.array([1., 1., -1.])
    res = opt.minimize(func, z, jac = gradient, method=adam)
    resx = np.zeros(3)
    resx[0] = res.x[0]
    resx[1] = res.x[1]*target_std
    resx[2] = res.x[2]*target_std + target_mean

    if method == 'W':
        quantiles = list(map(lambda y: quantile_weibull(res.x[0], res.x[1]*target_std, res.x[2]*targ
    elif method == 'LW':
        quantiles = list(map(lambda y: quantile_logweibull(res.x[0], res.x[1]*target_std, res.x[2]*t
    elif method == 'RV':
        quantiles = list(map(lambda x: quantile_of_maxima(-res.x[0], np.log(res.x[1]*target_std), 1,
    return quantiles, resx
################################################################################
def QuantLS(feature, target, method, level, N, alphas):

    target = np.array(target)
```

```
feature = np.array(feature)
mintarget = target.min() - math.e - 1
target -= mintarget
d1 = {'target': target, 'feature': feature}
data2 = pd.DataFrame(d1)

if method == 'W':
    m = smf.quantreg('np.log(target) ~ np.log(-np.log(feature))', data=data2)
    mfit = m.fit(q = level)
    quantiles = np.array(list(map(lambda x: quantile_weibull_trivial(mfit.params[1], np.exp(mfit
elif method == 'LW':
    m = smf.quantreg('np.log(np.log(target)) ~ np.log(-np.log(feature))', data=data2)
    mfit = m.fit(q = level)
    quantiles = np.array(list(map(lambda x: quantile_logweibull_trivial(mfit.params[1], np.exp(m
elif method == 'RV':
    m = smf.quantreg('np.log(target) ~ np.log(feature)', data=data2)
    mfit = m.fit(q = level)
    quantiles = np.array(list(map(lambda x: quantile_trivial(mfit.params[1], mfit.params[0], x,
```