

✓ Installing Dependancies

```
!pip install nelson_siegel_svensson
!pip install yfinance
```



Collecting nelson_siegel_svensson

Downloading nelson_siegel_svensson-0.5.0-py2.py3-none-any.whl.metadata (6.7 kB)

Requirement already satisfied: Click>=8.0 in /usr/local/lib/python3.11/dist-packages

Requirement already satisfied: numpy>=1.22 in /usr/local/lib/python3.11/dist-packages

Requirement already satisfied: scipy>=1.7 in /usr/local/lib/python3.11/dist-packages

Requirement already satisfied: matplotlib>=3.5 in /usr/local/lib/python3.11/dist-pack

Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-pac

Requirement already satisfied: cycycler>=0.10 in /usr/local/lib/python3.11/dist-package

Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-pa

Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-pa

Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-pack

Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (

Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-pac

Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist

Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (f

Downloading nelson_siegel_svensson-0.5.0-py2.py3-none-any.whl (9.9 kB)

Installing collected packages: nelson_siegel_svensson

Successfully installed nelson_siegel_svensson-0.5.0

Requirement already satisfied: yfinance in /usr/local/lib/python3.11/dist-packages (0

Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.11/dist-packag

Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.11/dist-packag

Requirement already satisfied: requests>=2.31 in /usr/local/lib/python3.11/dist-packa

Requirement already satisfied: multitasking>=0.0.7 in /usr/local/lib/python3.11/dist-

Requirement already satisfied: platformdirs>=2.0.0 in /usr/local/lib/python3.11/dist-

Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.11/dist-package

Requirement already satisfied: frozendict>=2.3.4 in /usr/local/lib/python3.11/dist-pa

Requirement already satisfied: peewee>=3.16.2 in /usr/local/lib/python3.11/dist-packa

Requirement already satisfied: beautifulsoup4>=4.11.1 in /usr/local/lib/python3.11/di

Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.11/dist-packag

Requirement already satisfied: typing-extensions>=4.0.0 in /usr/local/lib/python3.11/

Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/di

Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packa

Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/

Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-package

Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-p

Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-p

Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (f



```
import pandas as pd
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline
from nelson_siegel_svensson.calibrate import calibrate_ns_ols
import numpy as np
from numpy import linalg as LA
import seaborn as sns
```

✓ Task 2

✓ Q2. a. and Q2. b.

We selected the country - India. The following dataset contains the daily yields of the Indian bonds with the following maturities from 2006 to 2025(current).

Maturity	Source
3 Months	India 3-Month Bond Yield
6 Months	India 6-Month Bond Yield
1 Year	India 1-Year Bond Yield
2 Year	India 2-Year Bond Yield
3 Year	India 3-Year Bond Yield
4 Year	India 4-Year Bond Yield
5 Year	India 5-Year Bond Yield
6 Year	India 6-Year Bond Yield
7 Year	India 7-Year Bond Yield
8 Year	India 8-Year Bond Yield
9 Year	India 9-Year Bond Yield
10 Year	India 10-Year Bond Yield
12 Year	India 12-Year Bond Yield
15 Year	India 15-Year Bond Yield
24 Year	India 24-Year Bond Yield
30 Year	India 30-Year Bond Yield

```
df = pd.read_csv('C:/Users/Gast01/WQU-GWP1/India Bond Yield Data.csv') # Change the file
```

```
df.index = pd.to_datetime(df['Date'])
```

```
df = df.drop('Date', axis=1)
```

```
df = df.dropna()
```

```
def plot_std(df):
```

```
    """
```

```
    Plotting the standard deviation of the treasury yields for different maturities
```

```
    """
```

```
    y_std = df.std()
```

```
    fig, ax = plt.subplots()
```

```
    y_std.plot(figsize = (8,5),marker='o', title='Standard Deviations of Treasury Yields fo
```

```
    plt.xlabel("Maturity")
```

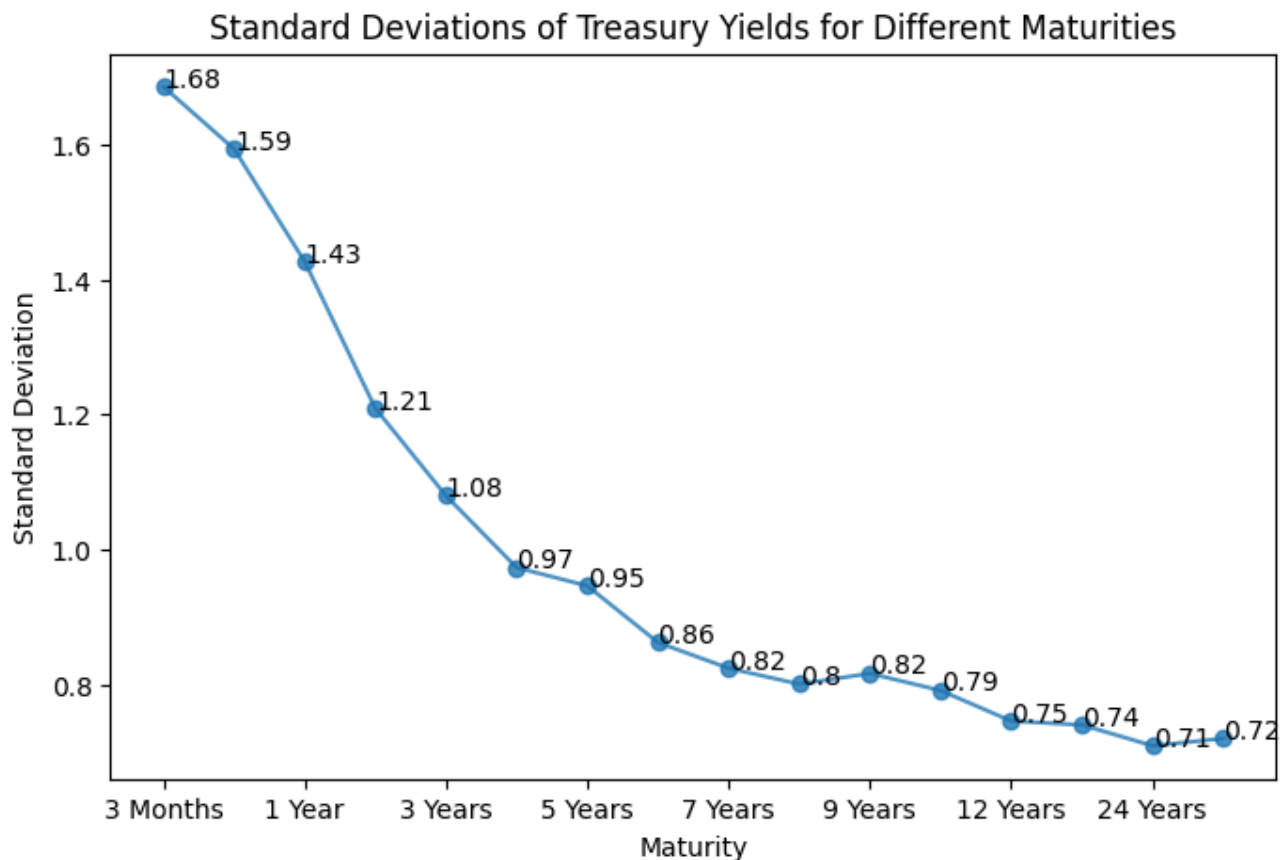
```
    plt.ylabel("Standard Deviation")
```

```
    for i in range(len(y_std)):
```

```
        ax.annotate(str(round(y_std.iloc[i],2)),xy=(i,y_std.iloc[i]))
```

```
    plt.show()
```

```
plot_std(df)
```



```
def plot_yield_curve(date, fig_n):
    """
    Plotting the yield curve for a given date
    """

    maturities = df.columns # Maturities
    fig, ax = plt.subplots(figsize=(6.15, 4))
    ax.plot(maturities, df.loc[date], marker='D', label='Yield Curve at ' + date)

    ax.set_yticklabels(['{:0.2f}%'.format(y) for y in ax.get_yticks()])
    ax.set_xticks(range(len(maturities)))
    labels = [m if i % 5 == 0 else '' for i, m in enumerate(maturities)]
    ax.set_xticklabels(labels)

    # Add labels and title
    ax.set_xlabel('Maturity')
    ax.set_ylabel('Yield')
    ax.set_title(fig_n+f'Treasury Yield Curve as of {date}')

    # Show the plot
    plt.grid(False)
    plt.show()
```

✓ Q2. c.

Fitting a Nelson Siegel Model on all the maturities on a specific date and comparing it with the original yield curve as of that date

```
def fit_ns(t, y, tau0=1.0):
    """
    Fitting the Nelson-Siegel model to the yield curve data for a given date.
    """

    curve, status = calibrate_ns_ols(t, y, tau0=1.0) # starting value of 1.0 for the optim
    assert status.success

    print(curve)

    return curve

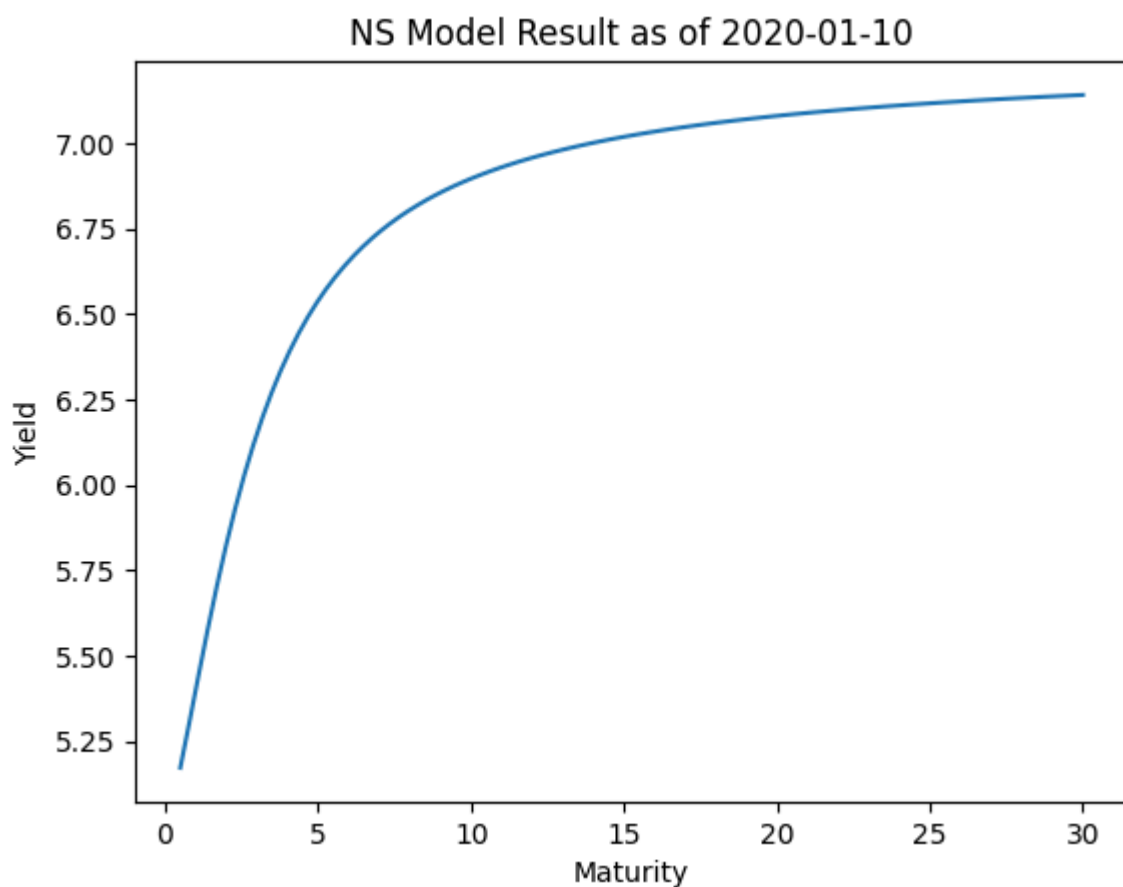
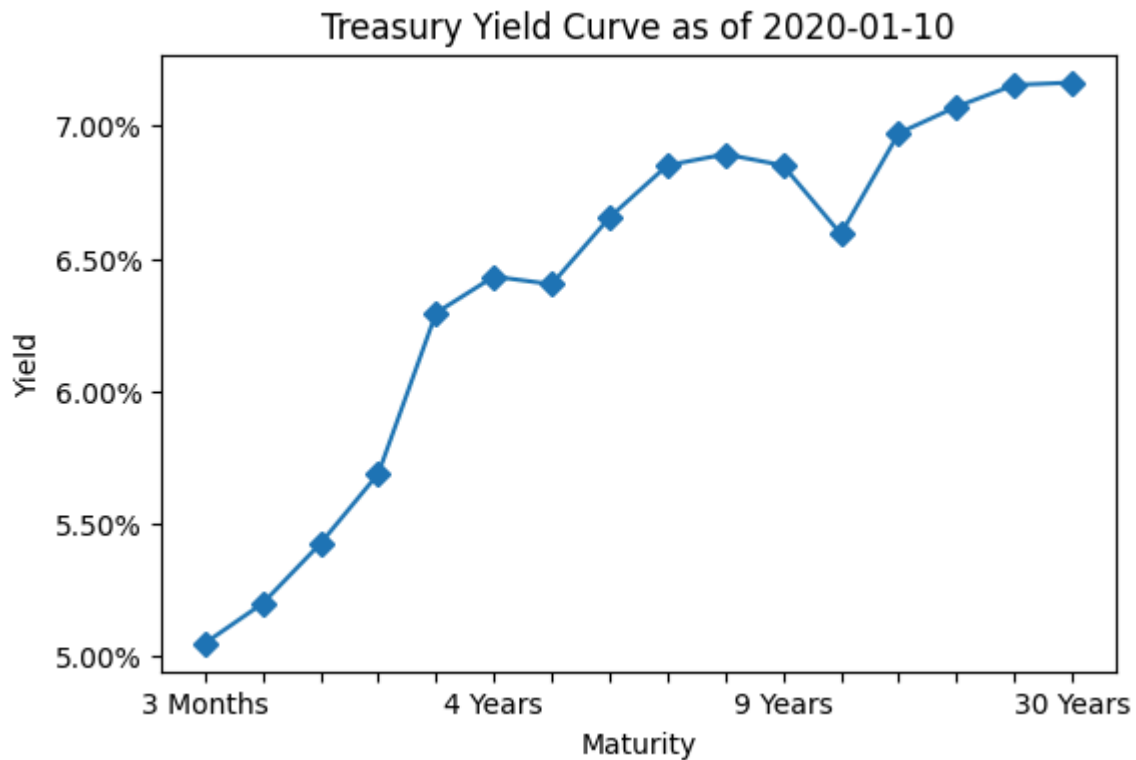

def plot_ns(date, y_hat, t_hat):
    """
    Plotting the yield curve for a given date.
    """

    plt.plot(t_hat, y_hat(t_hat))
    plt.xlabel("Maturity")
    plt.ylabel("Yield")
    plt.title(f"NS Model Result as of {date}")

    plt.show()


date = "2020-01-10"
t = np.array([0.25,0.5,1,2,3,4,5,6,7,8,9,10,12,15,24,30])
y = np.array(df.loc[date])
curve = fit_ns(t, y, tau0=1.0)
y_hat = curve
t_hat = np.linspace(0.5,30,100)
plot_yield_curve(date, '')
plot_ns(date, y_hat, t_hat)
```

```
↳ NelsonSiegelCurve(beta0=np.float64(7.262536953580489), beta1=np.float64(-2.2736710165  
<ipython-input-8-f6116200d263>:10: UserWarning: set_ticklabels() should only be used  
ax.set_yticklabels(['{:0.2f}%'.format(y) for y in ax.get_yticks()]])
```



Q2. f.

In the solution above, we have fit a Nelson Siegel Model on the yield curve as of 2020-01-10. The result `NelsonSiegelCurve(beta0=np.float64(7.262536960509667), beta1=np.float64(-2.273671047246356), beta2=np.float64(-1.779216868209846), tau=np.float64(0.9062504579008243))` indicates that the $\beta_0 = 7.26$ showing the level of the yield curve. $\beta_1 = -2.27$ shows the slope of the yield curve and the $\beta_2 = -1.77$ shows the shape or the curvature of the yield curve. the decay rate indicated by $\tau = 0.91$ showing a slow rate of decay.

A level of 7.26 shows that the long term expectation of the yields to be around 7%.

A negative slope can be attributed to the fact that there are dips in the long term yields, for example the 10 year yield is smaller than the 6 years, 7 years and 8 years yields.

A negative curvature indicates a concave behaviour. The yields started rising steep in the short run, but flatten over a longer period.

This interpretation is in line with the mid pandemic situation in 2020. Complete economic shutdowns has raised riskiness of the short term borrowings, hence the rise in yields sharply.

A smaller value of τ (close to 1) indicates that the effects of the slope and curvature parameters decay relatively slowly as maturity increases. This suggests that the short-to-medium-term rates have a strong influence on the shape of the curve, but as maturity increases, the curve flattens, and the level dominates.

✓ Q2. d. and Q2. f.

Fitting a Cubic Spline Model on all the maturities on a specific date and comparing it with the original yield curve as of that date.

Since there are 16 maturities, there would be 15 splines that pass through the 16 points. The splines can be shown as follows:

$$f(x) = a_1 x^3 + b_1 x^2 + c_1 x + d_1, \quad \text{when } 0.25 \leq x \leq 0.5$$

$$f(x) = a_2 x^3 + b_2 x^2 + c_2 x + d_2, \quad \text{when } 0.5 \leq x \leq 1$$

$$f(x) = a_3 x^3 + b_3 x^2 + c_3 x + d_3, \quad \text{when } 1 \leq x \leq 2$$

and so on...

From the above equations, we have ($15 \times 4 = 60$) unknowns. Hence, we need 60 equations to solve for the parameters.

Thus plugging each boundary, we get 30 equations as shown below:

$$a_1(0.25)^3 + b_1(0.25)^2 + c_1(0.25) + d_1 = 5.05 \quad (1)$$

$$a_1(0.5)^3 + b_1(0.5)^2 + c_1(0.5) + d_1 = 5.2 \quad (2)$$

$$\begin{aligned} a_2(0.5)^3 + b_2(0.5)^2 + c_2(0.5) + d_2 &= 5.2 \quad (3) \\ a_2(1)^3 + b_2(1)^2 + c_2(1) + d_2 &= 5.42 \quad (4) \end{aligned}$$

$$\begin{aligned} a_3(1)^3 + b_3(1)^2 + c_3(1) + d_3 &= 5.42 \quad (5) \\ a_3(2)^3 + b_3(2)^2 + c_3(2) + d_3 &= 5.69 \quad (6) \end{aligned}$$

Now since each interior point is a part of 2 splines, their slopes and curvatures must be the same. Therefore, their first order derivatives and their second order derivatives should be the same. Since there are (16 - 2 = 14) interior points, we get 14 first order equations, and 14 second order equations.

The first order equations are as follows:

$$\begin{aligned} 3a_1(0.5)^2 + 2b_1(0.5) + c_1 &= 3a_2(0.5)^2 + 2b_2(0.5) + c_2 \quad (31) \\ 3a_2(1)^2 + 2b_2(1) + c_2 &= 3a_3(1)^2 + 2b_3(1) + c_3 \quad (32) \end{aligned}$$

The second order equations are as follows:

$$\begin{aligned} 6a_1(0.5) + 2b_1 &= 6a_2(0.5) + 2b_2 \quad (45) \\ 6a_2(1) + 2b_2 &= 6a_3(1) + 2b_3 \quad (46) \end{aligned}$$

In total now we have 58 equations. Finally we assume the "Natural End Condition" and consider the second order derivatives of the exterior points to be 0/ Since there are 2 exterior points, we get 2 equations, which gives us full 60 equations.

$$\begin{aligned} 6a_1(0.25) + 2b_1 &= 0 \quad (59) \\ 6a_3(30) + 2b_3 &= 0 \quad (60) \end{aligned}$$

To calculate the cubic spline model, we use the `CubicSpline` function from the **SciPy** library, typically imported as `cs`.

The command is:

```
cs = CubicSpline(t, y, bc_type=((2, 0.0), (2, 0.0)))
```

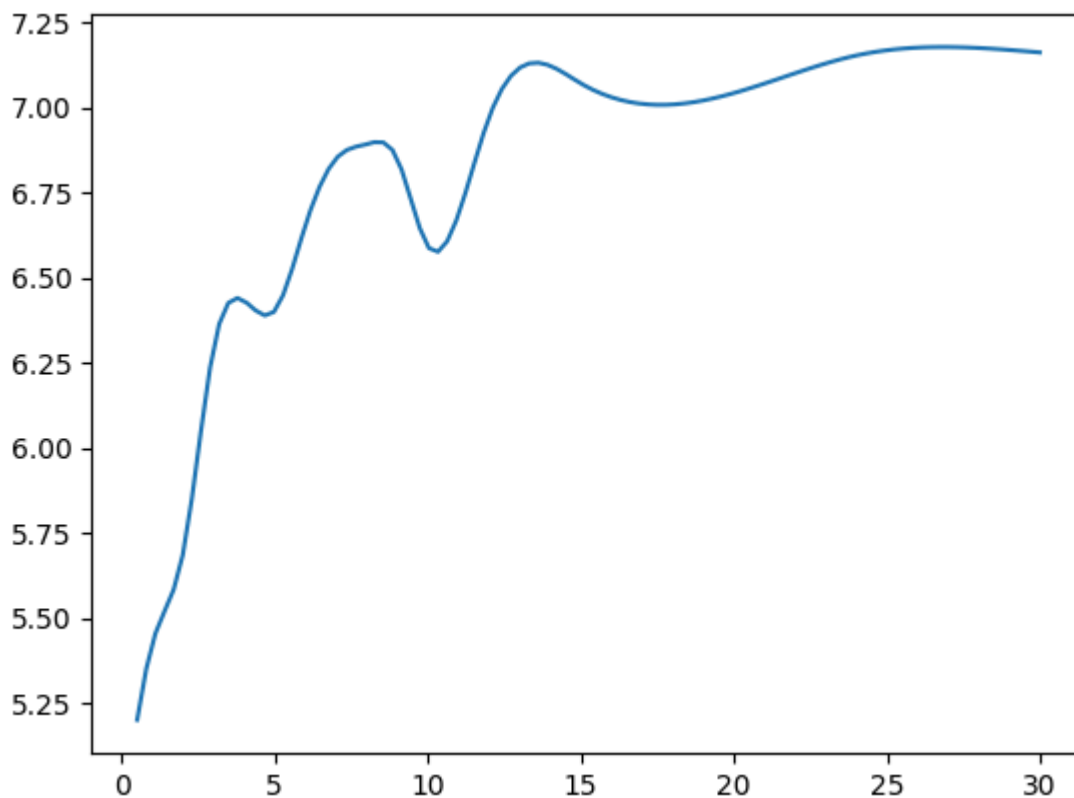
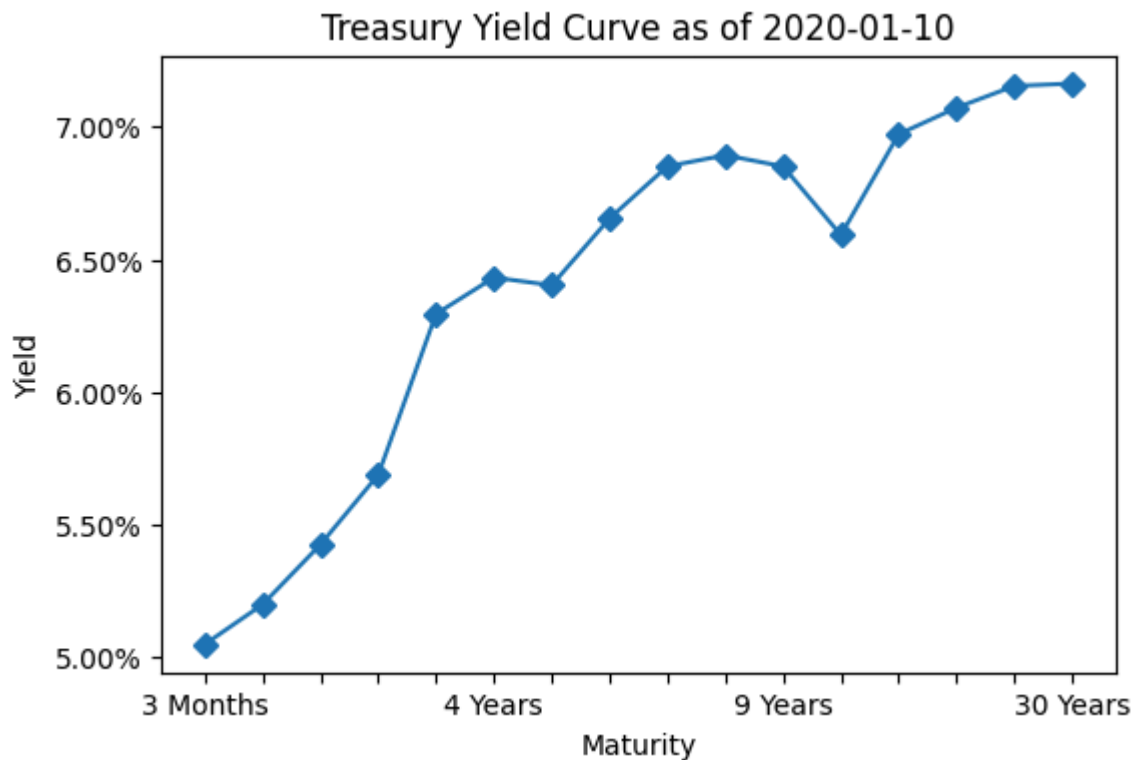
where:

- `t` = maturities
- `y` = yields as of a specific date
- `bc_type=((2, 0.0), (2, 0.0))` sets the **natural end condition** by specifying that the second derivative at the two endpoints is zero.

```
def fit_cs(t, y, t_hat):  
    """  
    Fitting a Cubic Spline Model on all the maturities on a specific date.  
    """  
    cs = CubicSpline(t, y, bc_type=((2, 0.0), (2, 0.0)))  
    interpolated_yields = cs(t_hat)  
    plt.plot(t_hat, interpolated_yields)  
  
date = "2020-01-10"  
t = np.array([0.25,0.5,1,2,3,4,5,6,7,8,9,10,12,15,24,30])  
y = np.array(df.loc[date])  
t_hat = np.linspace(0.5,30,100)  
plot_yield_curve(date, '')  
fit_cs(t, y, t_hat)
```



```
↳ <ipython-input-8-f6116200d263>:10: UserWarning: set_ticklabels() should only be used
    ax.set_yticklabels(['{:0.2f}%'.format(y) for y in ax.get_yticks()])
```



✓ Q2. e.

Fit:

- NS Model is a parametric model with defined parameters β_0 , β_1 and β_2 . The model smoothened the trend of the yield curve and just fit the general level with the slope and

curvature of the curve. It failed to capture the nuances of the mid term maturities.

Features like a lower yield in the 10 year maturity did not get captured in the NS Model.

- Cubic Spline is a non parametric model which fits a spline between each point individually and thus captures a much more nuanced view of the yield curve. The fit is indeed much better than the NS Model.

Interpretation:

- Since NS is a parametric approach, its parameters have a strict economic interpretation.
 - β_0 : Level
 - β_1 : Slope
 - β_2 : Curvature
 - τ : Decay Rate

It is helpful in making economic decision based on these parameters.

- Cubic Spline takes a non parametric approach and as a result does not have a direct economic interpretation. Its use cases are when one needs a data driven approach to mapping the yield movements, with a certain degree of smoothing involved. However, It is very susceptible to overfitting. Thus one must be careful towards the model complexity-overfitting tradeoff.

✓ Q2. g.

Indeed Smoothing data can be considered unethical, as discussed in M2 L4. However, whether or not the NS model smoothing is unethical, depends on the specific use cases.

NS Model is a parametric approach to smoothen out the yield curve based on its broad economic parameters, β_0 , β_1 , β_2 and τ . It aims to provide a simplified picture of the economic conditions based on the yields, to provide insights on the expected trends in the market both in the short and long term. In this case the smoothing out is not done to hide variability, it is done to cancel out the noise to identify trends. It is not unethical to smoothen the curve via NS Model in this case.

However, say in a situation where the user does the simplification, but fails to disclose it, with a purpose of hiding variability, or hiding a sharp downturn in the yields, it becomes unethical.

✓ Task 3

Q3. a & b) Generating 5 Gaussian uncorrelated random variables and running Principal Components on Covariance Matrix

```

np.random.seed(42)
sim_yld = np.random.normal(loc = 0, scale = 0.01, size = (200,5))
yld = pd.DataFrame(sim_yld, columns = ['3 Months', '1 Year', '3 Years', '10 Years', '30 Y
mean_yld = yld.mean()
std_yld = yld.std()
yc_standardised = (yld - mean_yld) / std_yld
std_data_cov = yc_standardised.cov()
eigenvalues, eigenvectors = LA.eig(std_data_cov)
principal_components = yc_standardised.dot(eigenvectors)
principal_components.columns = ["PC_1", "PC_2", "PC_3", "PC_4", "PC_5"]
df_eigval = pd.DataFrame({"Eigenvalues":eigenvalues}, index=range(1,6))
df_eigval["Explained proportion"] = df_eigval["Eigenvalues"] / np.sum(df_eigval["Eigenval
df_eigval.style.format({"Explained proportion": "{:.2%}"})

```



	Eigenvalues	Explained proportion
1	0.821674	16.43%
2	1.169212	23.38%
3	0.929294	18.59%
4	1.066514	21.33%
5	1.013305	20.27%

Q3. c) The above df shows the principal componentsw described by their eigen values and the amount of explanation of the variance they respectively do. As we can see that in case of random variables, which are uncorrelated, the 5 principal components more or less describe an equal proportion of the variance. This can be owed to the fact that the multivariate distributions are i.i.d across the observations.

Q3 d) Plotting the Scree plot of the variance explained by each component.

```
# Plotting the Scree plot
```

```

plt.figure(figsize=(8, 6))
plt.plot(range(1, 6), df_eigval['Eigenvalues'], marker='o', linestyle='-', color='b', lab
plt.title("Scree Plot")
plt.xlabel("Principal Component")
plt.ylabel("Eigenvalue")
plt.xticks(range(1, 6))

```

```
# Step 2: Plotting explained proportion as bars
```

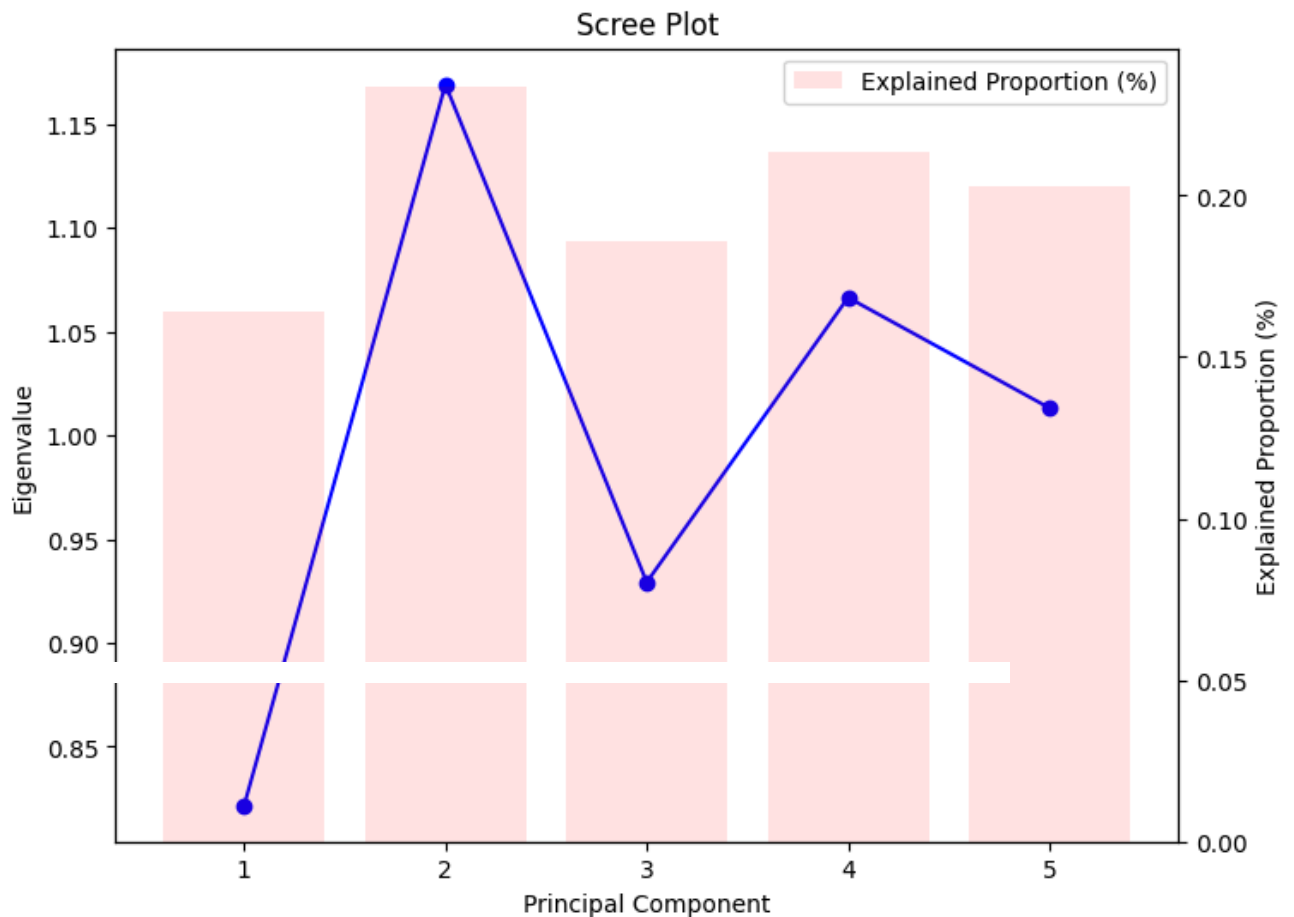
```

plt.twinx()
plt.bar(range(1, 6), df_eigval['Explained proportion'], alpha=0.1, color='r', label="Expl
plt.ylabel("Explained Proportion (%)")

plt.grid(False)
plt.legend(loc="upper right")

```

```
plt.show()
```



Here we can see that the explained variance does not reduce as the number of principal components increase. Infact they explain more or less similar proportions of the variance.

Q3. e & f) Loading actual data of government securities, for the past 6 months and converting yields into yield changes.

```
df = pd.read_csv('C:/Users/Gast01/WQU-GWP1/India Bond Yield Data.csv')
df.index = pd.to_datetime(df['Date'])
df_sec = df[df.index >= (df.index.max() - pd.DateOffset(months=6))]

# Selecting 5 securities
df_sec = df_sec[['3 Months', '1 Year', '3 Years', '10 Years', '30 Years']]

# Converting yields into yield changes and dropping NaNs
df_pct_chg = df_sec.pct_change().dropna()
```

Q3. g) Running Principal Components on original yields, using covariance matrix.

```

yld_chg_mean = df_pct_chg.mean()
yld_chg_std = df_pct_chg.std()
standardized_yld_chag = (df_pct_chg - yld_chg_mean) / yld_chg_std
std_yld_chg_cov = standardized_yld_chag.cov()
eigenvalues_yld, eigenvectors_yld = LA.eig(std_yld_chg_cov)
principal_components = standardized_yld_chag.dot(eigenvectors_yld)
principal_components.columns = ["PC_1", "PC_2", "PC_3", "PC_4", "PC_5"]
df_eigval_yld = pd.DataFrame({"Eigenvalues": eigenvalues_yld}, index=range(1,6))
df_eigval_yld["Explained proportion"] = df_eigval_yld["Eigenvalues"] / np.sum(df_eigval_y
df_eigval_yld.style.format({"Explained proportion": "{:.2%}"})

```



	Eigenvalues	Explained proportion
1	2.692920	53.86%
2	1.046973	20.94%
3	0.754758	15.10%
4	0.122989	2.46%
5	0.382360	7.65%

Q3. h) From the above df we can see a different picture regarding the eigenvectors and their explanation of the variance. The first PC explains more than 50% of the variance in the yields. This is the "Level" of the yield curve. This shows the long term expectations of the yields. The second PC explains almost 21% of the variance. We can interpret this as the "Slope" of the yield curve. Finally the 3rd PC explains about 15% of the variance, and this can be attributed to the "Curvature" of the yield curve, or how fast the slope is changing. The last 2 PCs explain relatively lower proportion of the variance, since the level, slope and curvature are the 3 main principal components.

Q3. i) Producing Scree Plot

Plotting the Scree plot

```

plt.figure(figsize=(8, 6))
plt.plot(range(1, 6), df_eigval_yld['Eigenvalues'], marker='o', linestyle='-', color='b',
plt.title("Scree Plot")
plt.xlabel("Principal Component")
plt.ylabel("Eigenvalue")
plt.xticks(range(1, 6))
plt.grid(False)

```

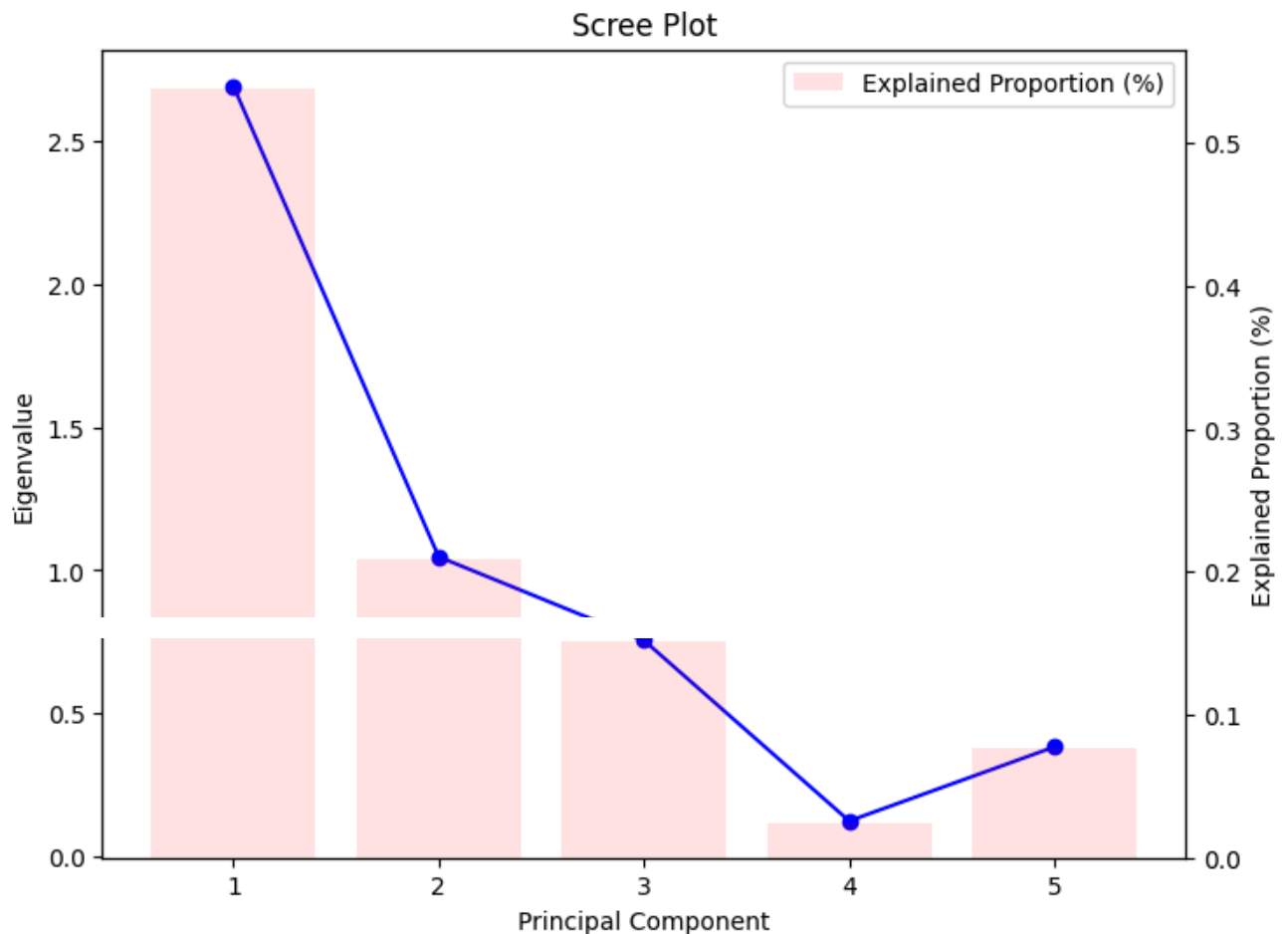
Plotting explained proportion as bars

```

plt.twinx()
plt.bar(range(1, 6), df_eigval_yld['Explained proportion'], alpha=0.1, color='r', label="
plt.ylabel("Explained Proportion (%)")
plt.legend(loc="upper right")

```

```
# Show the plot  
plt.show()
```



Q3. j.

The above scree plot is more in line with our expectations. The plot can be read as an elbow method. The elbow forming when the first principal component is explaining the highest variance and as the number of PCs increase, they explain lesser and lesser of the variance of the yield curve. In our case, the first three principal components explain about 92% of the data.

This observation is different than the uncorrelated gaussian random variables case because the data was generated such that the random variables had a same variance. As a result there is no dominant direction, and the variance is equally spread across the 5 eigenvalues, so the PC algorithm finds similar eigenvalues for the 5 PCs.

Since principal components are dimensionality reduction techniques, they work well when the variables are closely correlated. From the following figure we see that the yield changes from the actual data is more correlated than the gaussian data generation process. Hence PCA performed better on the real data.

```
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
```

```
# Simulated yield changes
```

```
sns.heatmap(yc_standardised.corr(), annot=True, cmap='coolwarm', fmt=".1f", ax=axes[0])
```

```
axes[0].set_title('Simulated Yield Changes')
```

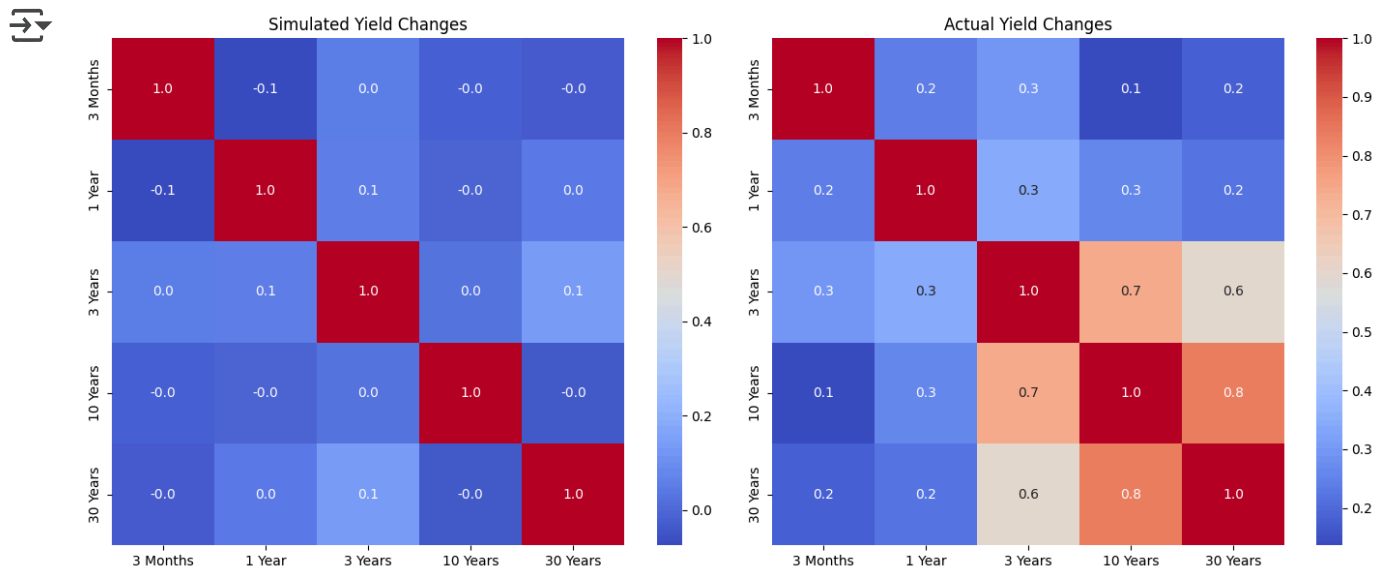
```
# Actual yield changes
```

```
sns.heatmap(standardized_yld_chag.corr(), annot=True, cmap='coolwarm', fmt=".1f", ax=axes
```

```
axes[1].set_title('Actual Yield Changes')
```

```
plt.tight_layout()
```

```
plt.show()
```



QUESTION 4

Empirical Analysis of ETFs

QUESTIONS 4(a):

ANSWER

For the group work, we decided to select a utility ETF called XLU, which has about 30 Holdings. The index includes securities of companies from the following industries: electric utilities; water utilities; multi-utilities; independent power and renewable electricity producers; and gas utilities. The fund is non-diversified.

```
# Importing required libraries

import datetime
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import yfinance as yfin
import math

from numpy import linalg as LA

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.decomposition import TruncatedSVD

from datetime import date

pd.options.display.float_format = "{:,.6f}".format
```

QUESTION 4(b): Get at least 6 months of data (~ 120 data points).

One year of xlu data had been fetched from yfinance database as follows:

```
# Starting and end dates
start = datetime.date(2019, 1, 1)
end = datetime.date(2020, 1, 1)

xlu_tickers = [
    "NEE",    # NextEra Energy Inc.
    "SO",     # Southern Company
    "DUK",    # Duke Energy Corporation
    "AEP",    # American Electric Power Company Inc.
    "D",      # Dominion Energy Inc.
    "SRE",    # Sempra
    "EXC",    # Exelon Corporation
    "VST",    # Vistra Corp.
    "PEG",    # Public Service Enterprise Group Inc.
    "PCG",    # PG&E Corporation
    "XEL",    # Xcel Energy Inc.
    "ED",     # Consolidated Edison Inc.
    "AWK",    # American Water Works Company Inc.
    "WEC",    # WEC Energy Group Inc.
    "EIX",    # Edison International
    "ES",     # Eversource Energy
    "ATO",    # Atmos Energy Corporation
    "CMS",    # CMS Energy Corporation
```



```

"NI",    # NiSource Inc.
"PNW",    # Pinnacle West Capital Corporation
"CNP",    # CenterPoint Energy Inc.
"EVERG",  # Evergy Inc.
"FE",     # FirstEnergy Corp.
"NRG",    # NRG Energy Inc.
"OGE",    # OGE Energy Corp.
"AEE",    # Ameren Corporation
"AES",    # The AES Corporation
"LNT",    # Alliant Energy Corporation
"UGI",    # UGI Corporation
"IDA"     # IDACORP Inc.

```

```
]
```

```
# Get ETF data
```

```
df = yfin.download(xlu_tickers, start, end, auto_adjust = False)["Adj Close"]
```

```
# Convert DataFrame index to timezone-aware (UTC)
```

```
df.index = df.index.tz_localize('UTC')
```

 [*****100%*****] 30 of 30 completed

Let us have a look at the first five rows of the daily data.

```
df.head(5)
```



	Ticker	AEE	AEP	AES	ATO	AWK	CMS	C
Date								
2019-01-02 00:00:00+00:00		53.270641	58.418915	11.534038	77.131935	79.253159	40.172016	23.2553
2019-01-03 00:00:00+00:00		53.404797	58.282764	11.525901	77.715683	79.565666	40.246845	23.3881
2019-01-04 00:00:00+00:00		54.159466	58.819359	11.908203	78.711479	80.163826	40.621037	23.8200
2019-01-07 00:00:00+00:00		53.538967	58.490982	11.965139	78.136322	79.315689	40.413155	23.9114
2019-01-08 00:00:00+00:00		54.385860	59.211826	12.241695	79.157890	80.476295	41.020195	24.3682


5 rows × 30 columns



Overview of the ETF Data

We can use the pandas *describe()* method to show summary stats of our data. We can see that all assets have same number of observations (count) since they all belong to the same portfolio. The other summary stats are relatively basic, like mean and standard deviation along with showing minimum, maximum, and a few quantiles.

```
df.describe()
```

 Ticker	AEE	AEP	AES	ATO	AWK	CMS	C
count	252.000000	252.000000	252.000000	252.000000	252.000000	252.000000	252.0000
mean	62.849828	71.180582	14.031441	90.896079	101.687287	49.038936	24.5471
std	3.245782	5.109215	1.068596	5.416583	9.813963	3.909191	1.1966
min	53.270641	58.282764	11.525901	77.131935	79.253159	40.172016	21.0997
25%	60.837073	67.466379	13.314819	87.135109	93.918928	45.967123	24.0724
50%	63.895060	72.929676	13.999063	92.650085	104.843285	49.257168	24.7338
75%	65.094200	75.268436	14.701836	95.203949	110.105621	52.592816	25.5131
max	68.704857	78.527794	16.838137	99.922478	117.211868	55.235107	26.3220

8 rows × 30 columns



QUESTION 4(C) :

ANSWER Importance of Daily Returns

In financial analysis, the daily return of an asset, XLU-ETF in our case, is a fundamental metric used to measure its performance over a single trading day. It definitely aids in quantifying the percentage change in the asset's value between the close of one trading day and the close of the subsequent day. To compute a daily return you have to use the closing prices of the asset on two consecutive trading days.

To account for dividend and stock splits, technically, one must use adjusted closing price to avoid impact on the individual prices of an ETF constituents assets/holdings.

```
# 1. Compute daily log returns
daily_returns = df.pct_change() # Used simple return to compute the daily return (alternatively, you could use adjusted closing prices)
daily_returns = daily_returns.dropna()
daily_returns.head()
```



Ticker	AEE	AEP	AES	ATO	AWK	CMS	CNP
Date							
2019-01-03 00:00:00+00:00	0.002518	-0.002331	-0.000705	0.007568	0.003943	0.001863	0.005714
2019-01-04 00:00:00+00:00	0.014131	0.009207	0.033169	0.012813	0.007518	0.009297	0.018466
2019-01-07 00:00:00+00:00	-0.011457	-0.005583	0.004781	-0.007307	-0.010580	-0.005118	0.003835
2019-01-08 00:00:00+00:00	0.015818	0.012324	0.023113	0.013074	0.014633	0.015021	0.019104
2019-01-09 00:00:00+00:00	-0.007401	-0.007575	-0.001993	-0.016376	-0.012758	-0.009528	-0.012270

5 rows × 30 columns

**Question 4(d): Compute the covariance matrix.**

Before standardizing the data, you need to calculate the daily returns of the stocks in the XLU ETF, which is typically done using the percentage change between consecutive closing prices. The covariance matrix is calculated using the daily returns, not the raw prices. It measures how the returns of different XLU ETF stocks move together.

Without calculating the daily returns first, the standardization wouldn't make sense. In fact, standardization makes varying swings (big and small) comparable by adjusting them so they all have an average change of 0 and typical change of 1. This is helpful for analysis where the scale of the data matters.

```
# Standardize stock returns dataset
daily_returns_means = daily_returns.mean()
daily_returns_stds = daily_returns.std()
standardized_returns = (daily_returns - daily_returns_means) / daily_returns_stds
standardized_returns.head()
```



Ticker	AEE	AEP	AES	ATO	AWK	CMS	CNP
Date							
2019-01-03 00:00:00+00:00	0.183450	-0.417105	-0.180855	0.763051	0.275034	0.074791	0.474443
2019-01-04 00:00:00+00:00	1.498006	0.948051	2.525074	1.371657	0.663016	0.908490	1.557158
2019-01-07 00:00:00+00:00	-1.398540	-0.801923	0.257431	-0.962968	-1.301269	-0.707954	0.314894
2019-01-08 00:00:00+00:00	1.688990	1.316904	1.721830	1.401925	1.435249	1.550297	1.611336
2019-01-09 00:00:00+00:00	-0.939365	-1.037670	-0.283738	-2.015199	-1.537639	-1.202506	-1.052580

5 rows × 30 columns



```
# Calculate covariance for standardized return matrix
standardized_returns_dvd_sqrt_n=(standardized_returns/math.sqrt(len(standardized_returns))
standardized_returns_cov = standardized_returns_dvd_sqrt_n.T@standardized_returns_dvd_sqrt_n
standardized_returns_cov.head()
```

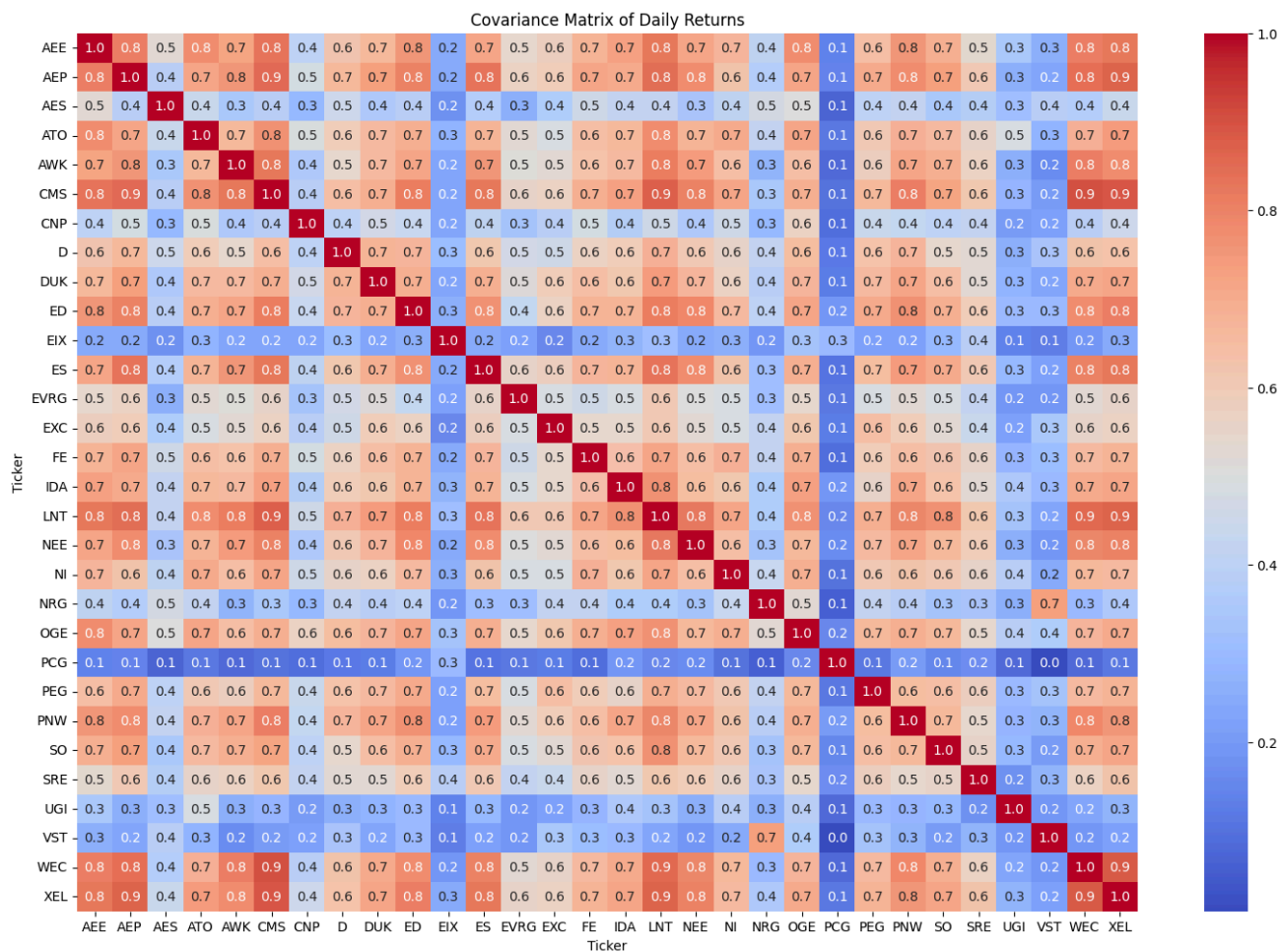


Ticker	AEE	AEP	AES	ATO	AWK	CMS	CNP	D
Ticker								
AEE	1.000000	0.806508	0.522383	0.771703	0.724853	0.834286	0.400582	0.620391
AEP	0.806508	1.000000	0.389617	0.716671	0.757821	0.851061	0.479604	0.661853
AES	0.522383	0.389617	1.000000	0.440487	0.343414	0.386723	0.291766	0.451335
ATO	0.771703	0.716671	0.440487	1.000000	0.665880	0.763396	0.497985	0.592841
AWK	0.724853	0.757821	0.343414	0.665880	1.000000	0.827253	0.360149	0.540264

5 rows × 30 columns



```
plt.figure(figsize=(18, 12))
sns.heatmap(standardized_returns_cov, annot=True, cmap='coolwarm', fmt=".1f")
plt.title('Covariance Matrix of Daily Returns')
plt.show()
```



Question 4(e):

(compare and contrast PCA and SVD, explain what the eigenvectors, eigenvalues, singular values etc show us for the specific data, etc,)

ANSWER

Both PCA and SVD are dimensionality reduction technique. They reduce dimensions of matrices but importantly they do retain substantial information or rather variance of a given financial data.

PCA plays a critical role of identifying key factors behind asset price movements whilst reducing dimensionality of risk-based models. Through PCA, one can establish a diversified portfolio based on the principal components. PCA can keep track of unique patterns and ambiguities in portfolio/ETF, which might not be captured in individual asset.

SVD stands for Singular Value Decomposition. It is a robust factorization technique that can effectively decompose any given matrix (Symmetric or not) into 3 matrices, namely, U, S and V^T . These matrices have $m \times n$, $m \times n$ and $n \times n$ dimensions respectively. The magnitude of each singular value shows the importance of the corresponding dimension in the data.

Eigen vectors are extracted from PCA and they do represent the principal components which depict relative movements in daily returns of assets. On the other hand, eigenvalues, provides the amount of variance in the daily returns data related to principal components (eigenvector). The higher the eigenvalue, the better it gives information about the asset under analysis.

The following code solutions, shows how Eigenvalues and Eigenvectors help generate Principal Components based on ULX-ETF data under analysis. It is important to note that SVD can be applied to generate eigenvalues and eigenvectors.

```
# Calculate eigenvectors and eigenvalues of the covariance matrix of standardized dataset
eigenvalues, eigenvectors = np.linalg.eig(standardized_returns_cov)
eigenvalues
```

```
array([[16.94028088,  1.84467656,  1.26660112,  1.0033836 ,  0.85284607,
         0.78526804,  0.68313883,  0.62148367,  0.56747695,  0.54817649,
         0.50067321,  0.4355354 ,  0.39437057,  0.06999438,  0.08276568,
         0.09744171,  0.10123079,  0.13109828,  0.36516273,  0.35232383,
         0.33001439,  0.29732338,  0.28207821,  0.16274534,  0.17443758,
         0.19353803,  0.24393001,  0.23422616,  0.21796549,  0.21981263])
```

```
print(pd.DataFrame(eigenvectors).head())
```

```

0 0.213975  0.029617  0.048260 -0.097896  0.130891  0.023761  0.042475
1 0.219064  0.116065  0.034320  0.081349  0.006026 -0.016249  0.048878
2 0.127139 -0.328994  0.058621 -0.160035  0.144884  0.216879  0.387771
3 0.202651 -0.031364 -0.007038 -0.274620 -0.072991  0.025539 -0.076439
4 0.201148  0.183299  0.061812 -0.008566  0.061080  0.105053 -0.072025

7 0.207650  0.096307  0.051588 ... -0.028883  0.341933 -0.220457 -0.238741
1 0.026485 -0.012477  0.066108 ...  0.192387  0.086952 -0.160607  0.223562
```

```

2 -0.658986 -0.141501 -0.186631 ... -0.012539 -0.139527 0.114488 0.021493
3 0.032491 0.134994 0.117323 ... -0.166482 0.374353 -0.255861 0.304626
4 0.013562 0.152696 -0.084538 ... -0.322136 -0.388158 0.055149 0.373392

      24      25      26      27      28      29
0 -0.037419 0.028039 -0.020517 -0.062485 0.291977 0.005230
1 0.065085 -0.054074 0.125435 0.118862 0.127297 0.178015
2 -0.085401 -0.007385 0.002403 -0.010940 -0.185881 -0.053974
3 0.157672 0.122336 -0.450374 -0.105967 -0.379816 -0.083053
4 -0.050213 0.051465 -0.031854 -0.431146 0.331905 -0.270027

```

[5 rows x 30 columns]

```

# Transform standardized data with Loadings
principal_components = standardized_returns_cov.dot(eigenvectors)
principal_components.columns = ["PC_" + str(i) for i in range(1, 31)]
principal_components.head()

```



	PC_1	PC_2	PC_3	PC_4	PC_5	PC_6	PC_7	PC_8
Ticker								
AEE	3.624796	0.054633	0.061126	-0.098228	0.111630	0.018659	0.029016	-0.129016
AEP	3.710998	0.214102	0.043469	0.081624	0.005139	-0.012760	0.033391	0.016410
AES	2.153771	-0.606888	0.074249	-0.160577	0.123564	0.170308	0.264902	-0.409500
ATO	3.432957	-0.057857	-0.008914	-0.275549	-0.062250	0.020055	-0.052218	0.020110
AWK	3.407511	0.338127	0.078291	-0.008595	0.052092	0.082494	-0.049203	0.008410

5 rows x 30 columns



Understanding the explained proportion helps us determine the relative importance of each principal component. By selecting the top principal components with the highest explained proportions, we can reduce the dimensionality of the data while retaining most of the important information.

We used the following codes to demonstrate how to organize the calculated eigenvalues and their corresponding explained proportions into a pandas DataFrame for easier analysis.

Explained Proportion of 56.47% means that this 'PC_1' alone captures or explains 56.47% of the total variance observed in the standardized daily returns of the XLU ETF's assets.

```

# Put data into a DataFrame
df_eigval = pd.DataFrame({"Eigenvalues":eigenvalues}, index=range(1,31))

# Work out explained proportion
df_eigval["Explained proportion"] = df_eigval["Eigenvalues"] / np.sum(df_eigval["Eigenval
#Format as percentage
df_eigval.style.format({"Explained proportion": "{:.2%}"})

```



	Eigenvalues	Explained proportion
1	16.940281	56.47%
2	1.844677	6.15%
3	1.266601	4.22%
4	1.003384	3.34%
5	0.852846	2.84%
6	0.785268	2.62%
7	0.683139	2.28%
8	0.621484	2.07%
9	0.567477	1.89%
10	0.548176	1.83%
11	0.500673	1.67%
12	0.435535	1.45%
13	0.394371	1.31%
14	0.069994	0.23%
15	0.082766	0.28%
16	0.097442	0.32%
17	0.101231	0.34%
18	0.131098	0.44%
19	0.365163	1.22%
20	0.352324	1.17%
21	0.330014	1.10%
22	0.297323	0.99%
23	0.282078	0.94%
24	0.162745	0.54%
25	0.174438	0.58%
26	0.193538	0.65%
27	0.243930	0.81%
28	0.234226	0.78%
29	0.217965	0.73%
30	0.219813	0.73%


```
# Visualization for PCA
```

```
x = standardized_returns_cov.values # Convert DataFrame to NumPy array  
x = StandardScaler().fit_transform(x) # Standardize the data
```

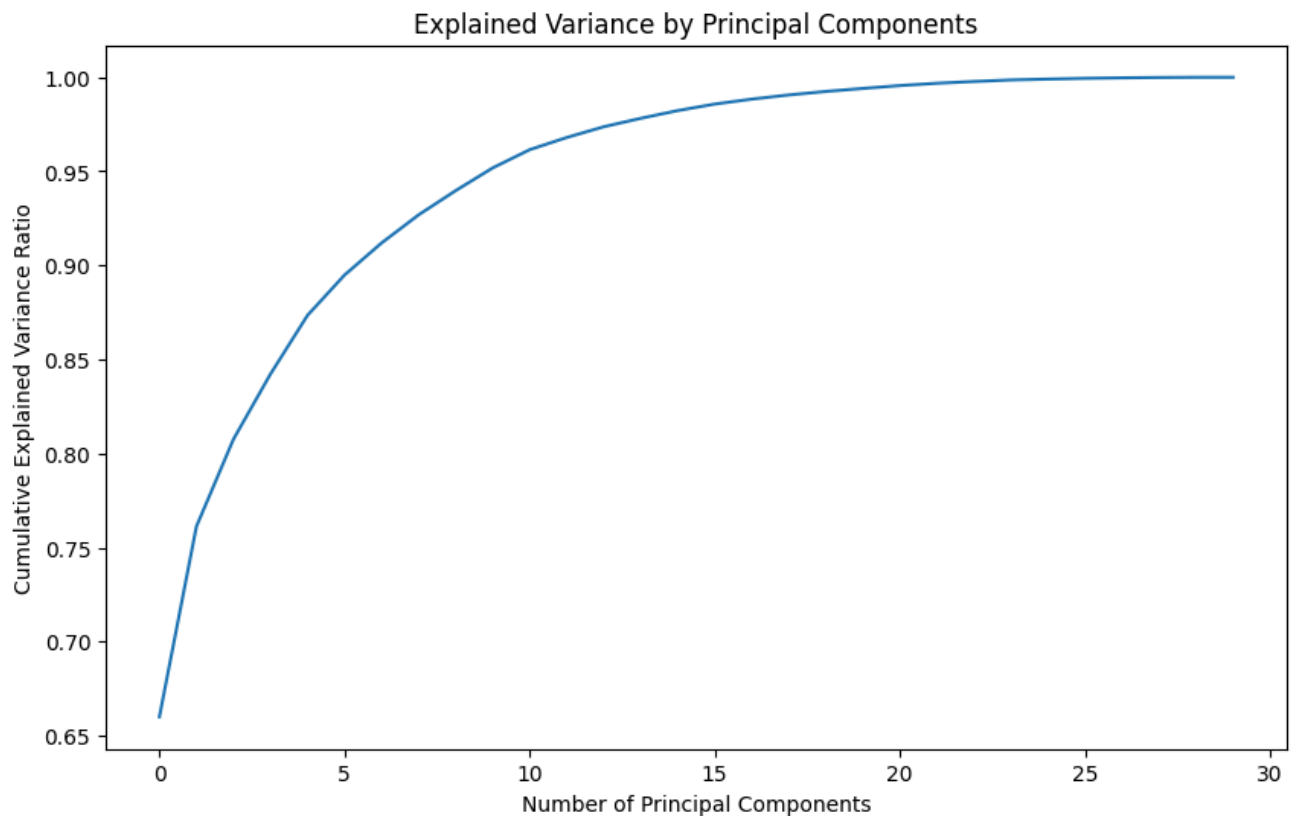
```
# Apply PCA
```

```
pca = PCA(n_components=standardized_returns_cov.shape[1])  
principalComponents = pca.fit_transform(x)  
principalDf = pd.DataFrame(data=principalComponents, columns=['principal component ' + st
```

```
# Get explained variance ratio
```

```
explained_variance_ratio = pca.explained_variance_ratio_
```

```
plt.figure(figsize=(10, 6))  
plt.plot(np.cumsum(explained_variance_ratio))  
plt.xlabel('Number of Principal Components')  
plt.ylabel('Cumulative Explained Variance Ratio')  
plt.title('Explained Variance by Principal Components')  
plt.show()
```



Question 4(f): Compute the SVD.

The task here involves decomposing the daily returns matrix of the XLU ETF's assets using Singular Value Decomposition (SVD). SVD is a powerful technique for breaking down a matrix into its fundamental components, revealing underlying relationships and patterns within the data.

By applying SVD to the daily returns matrix, we aim to identify the most significant dimensions or factors that drive the variability in the returns of the XLU ETF's assets. Secondly, gain insights into the relationships between different assets within the ETF and how they contribute to overall portfolio risk and return and transform the original data into a more manageable form, making it

```
# Perform SVD for stock returns
U, s, VT = np.linalg.svd(daily_returns)

# Use SVD to calculate eigenvectors and eigenvalues of the covariance matrix of standardized returns
U_st_return, s_st_return, VT_st_return = np.linalg.svd(standardized_returns_dvd_sqrt_n)
print("\nSquared Singular values (eigenvalues):")
print(s_st_return**2)
print("\nMatrix V (eigenvectors)")
print(pd.DataFrame(VT_st_return.T).head())
```



Squared Singular values (eigenvalues):

```
[16.94028088  1.84467656  1.26660112  1.0033836  0.85284607  0.78526804
  0.68313883  0.62148367  0.56747695  0.54817649  0.50067321  0.4355354
  0.39437057  0.36516273  0.35232383  0.33001439  0.29732338  0.28207821
  0.24393001  0.23422616  0.21981263  0.21796549  0.19353803  0.17443758
  0.16274534  0.13109828  0.10123079  0.09744171  0.08276568  0.06999438]
```

Matrix V (eigenvectors)

```
      0      1      2      3      4      5      6  \
0 -0.213975 -0.029617  0.048260 -0.097896  0.130891 -0.023761  0.042475
1 -0.219064 -0.116065  0.034320  0.081349  0.006026  0.016249  0.048878
2 -0.127139  0.328994  0.058621 -0.160035  0.144884 -0.216879  0.387771
3 -0.202651  0.031364 -0.007038 -0.274620 -0.072991 -0.025539 -0.076439
4 -0.201148 -0.183299  0.061812 -0.008566  0.061080 -0.105053 -0.072025

      7      8      9  ...    20      21      22      23  \
0 -0.207650  0.096307  0.051588  ... -0.005230 -0.291977  0.028039  0.037419
```