GROUP WORK PROJECT # ___ MScFE 600: FINANCIAL DATA

GROUP NUMBER: __8049_____

FULL LEGAL NAME	LOCATION (COUNTRY)	EMAIL ADDRESS	MARK X FOR ANY NON-CONTRIBUTING MEMBER
Jiayang Li	United Kingdom	jli68666666@gmail.com	
James Asira	Kenya	jjasira2016@gmail.com	
Dev Sandipkumar Bodiwala	India	devexbodiwala@gmail.com	

Statement of integrity: By typing the names of all group members in the text boxes below, you confirm that the assignment submitted is original work produced by the group (excluding any non-contributing members identified with an "X" above).

Team member 1	Jiayang Li
Team member 2	James Asira
Team member 3	Dev Sandipkumar Bodiwala

Use the box below to explain any attempts to reach out to a non-contributing member. Type (N/A) if all members contributed.
Note: You may be required to provide proof of your outreach to non-contributing members upon request.

Task 1:

a. The following is an example of poor quality structured data (only show the first five rows for the data)

CustomerID	Name	Age	Email	Purchase Amount	Purchase Date
1	Jack Frost	23	jack@gmail.com	120	2024-08-19
2		35	joe@gmail.com	130	2024-12-24
3	John Smith	11	john@yahoo.com	-50	2024/6/17
4	Michael Li	-26	michael@gmail.com	200	2024-06-25
5	Jack Frost	NULL	jack@gmail.com	120	2024-08-19

b. The data has poor quality because:

1. It has missing values. For example, Name is missing for CustomerID 2 and Age is NULL for CustomerID 5.

2. It has invalid values. For example, Purchase Amount is negative(-50) for CustomerID 3 and Age is negative(-26) for CustomerID 4.

3. It has different formatting for some data. For example, Purchase Date is 2024/6/17 for CustomerID 3 while other Purchase Dates follow 'year-month-day' format.

4. It has duplicate data. For example, CustomerID 1 and 5 have the same Name, Email, Purchase Amount and Purchase Date.

c. The following product review data is an example of poor quality unstructured data.

1. OMG this phone is waste of $$$$$$! battery dies <2hrs!! bad build quality why u sell this?? :((

2. !!!DO NOT BUY!!! I ordered dis last week & it came broke????? Customer service sucks.

3. Good prodct but delivery SUX waited 10 days and got wrong clr. fix YOUR shipping process! :/

4. i tink its good phone FOR price bt cam quality not as expected.

5. .....Totally great....phone.....best buy!! wow camera amazing battery rocks...💯💯💯.....

6. received as gift dnt like bcz itz android wanted apple. Returning. not good 4 me lolol.

d.

1.The data contains wrong spellings and informal language. For example, it uses words like “dis, tink, bcz” and it also includes informal phrases like "lolol"

2.The data contains symbols and emojis such as “$$$$$$”and “💯”. This can confuse natural language processing tools if we use this data for sentiment analysis.

3.The data contains inconsistent grammar and syntax such as "battery dies <2hrs!! bad build quality why u sell this??”.

4. The data contains ambiguity and missing context in some of the reviews. For example, reviews like "Totally great....phone" does not specify what exactly is great about the phone.

Task 2:

a. We pick United States Treasury securities (U.S. Treasuries).

b. We pick maturities ranging from 6 months to 30 years. The following is how the data looks like:

[6]:
```
def get_yield_data(series_id):
    data = fred.get_series(series_id, observation_start="2020-01-01", observation_end="2024-05-03")
    return data

yields_d = {series_id: get_yield_data(series_id) for series_id in series_ids}
yields = pd.DataFrame(yields_d)
yields.columns = ['6 Month', '1 Year', '3 Year', '5 Year', \
                  '10 Year', '20 Year', '30 Year']
yields.index = pd.to_datetime(yields.index)
yields = yields.dropna()
yields
```

[6]:

	6 Month	1 Year	3 Year	5 Year	10 Year	20 Year	30 Year
2020-01-02	1.57	1.56	1.59	1.67	1.88	2.19	2.33
2020-01-03	1.55	1.55	1.54	1.59	1.80	2.11	2.26
2020-01-06	1.56	1.54	1.56	1.61	1.81	2.13	2.28
2020-01-07	1.56	1.53	1.55	1.62	1.83	2.16	2.31
2020-01-08	1.56	1.55	1.61	1.67	1.87	2.21	2.35
...	...	...	...	...	...	...	...
2024-04-29	5.43	5.20	4.80	4.65	4.63	4.86	4.75
2024-04-30	5.44	5.25	4.87	4.72	4.69	4.90	4.79
2024-05-01	5.43	5.21	4.79	4.64	4.63	4.85	4.74
2024-05-02	5.42	5.16	4.71	4.57	4.58	4.82	4.72
2024-05-03	5.41	5.12	4.63	4.48	4.50	4.75	4.66

1087 rows × 7 columns

c. We pick yields on 2024-05-03 to fit a Nelson-Siegel model.

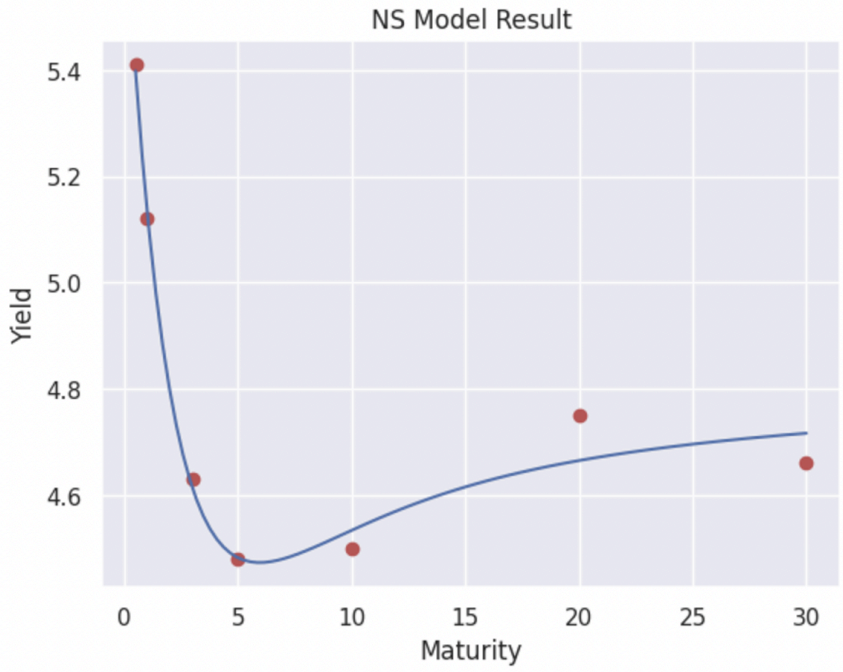
[8]:
```
from nelson_siegel_svensson.calibrate import calibrate_ns_ols
```

[11]:
```
t = np.array([0.5,1,3,5,10,20,30])
y = np.array(yields.loc["2024-05-03"])
```

[15]:
```
curve, status = calibrate_ns_ols(t, y, tau0=1.0)
assert status.success
y_est = curve
t_est = np.linspace(0.5,30,100)
plt.plot(t, y, 'ro')
plt.plot(t_est, y_est(t_est))
plt.xlabel("Maturity")
plt.ylabel("Yield")
plt.title("NS Model Result")
```

[15]:
```
Text(0.5, 1.0, 'NS Model Result')
```

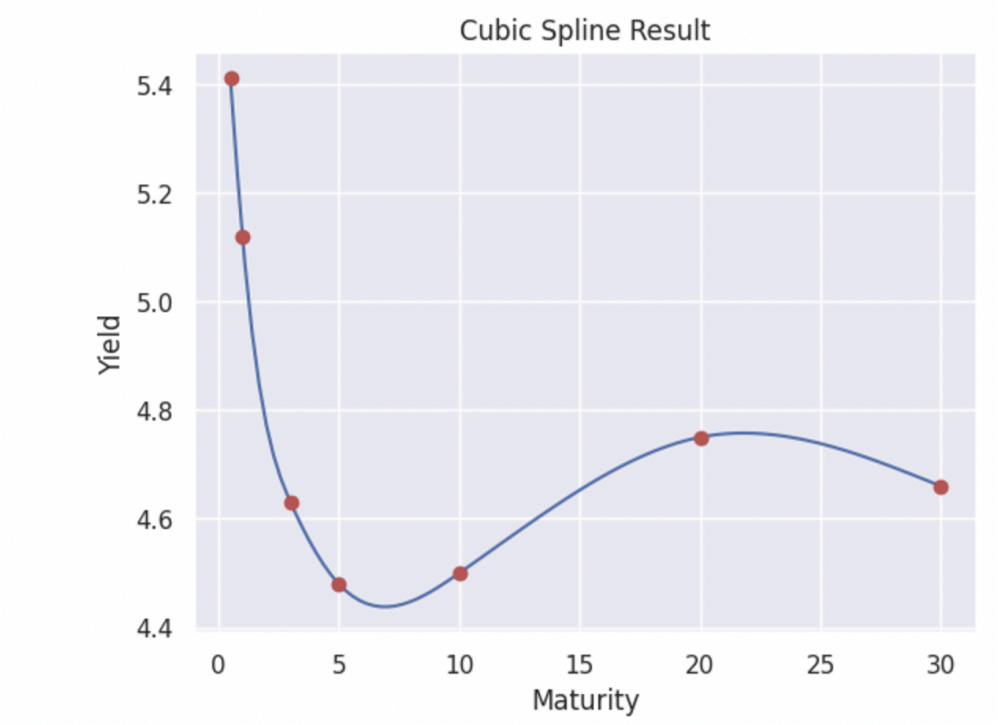
[15]: Text(0.5, 1.0, 'NS Model Result')



d. We pick yields on 2024-05-03 to fit a Cubic-Spline model.

```
[14]: from scipy.interpolate import CubicSpline
      t = np.array([0.5,1,3,5,10,20,30])
      y = np.array(yields.loc["2024-05-03"])
      f = CubicSpline(t, y, bc_type='natural')
      t_new = np.linspace(0.5,30,100)
      y_new = f(t_new)
      plt.plot(t_new, y_new)
      plt.plot(t, y, 'ro')
      plt.xlabel("Maturity")
      plt.ylabel("Yield")
      plt.title("Cubic Spline Result")
```

```
[14]: Text(0.5, 1.0, 'Cubic Spline Result')
```



e, f. We can see that both Nelson-Siegel model and Cubic-Spline model fit the data quite well. They both show the overall trend of first decreasing and then increasing. The Cubic-Spline model is more accurate and shows the last decreasing trend as well while the Nelson-Siegel model fails to capture this. This might be due to the fact that the Nelson-Siegel model is constrained by the exponential-decay terms, which might potentially lead to poor fit at extreme maturities. Since the Cubic-Spline model is a non-parametric, interpolation-based approach that fits piecewise cubic polynomials to the yield curve, it can fit this data and other highly irregular and complex yield curves well.

Another noticeable difference is that the Nelson-Siegel model does not pass all the points while the Cubic-Spline model passes all the points. This shows the Cubic-Spline model might have the risk of overfitting when we use too many knots in the Cubic-Spline model (which may exhibit unrealistic wiggles) and we need to adjust the number and placement of knots carefully. The

number of knots selected affects the bias-variance tradeoff: as the number of knots increases, the model risks overfitting the data, but too few knots can produce a more restrictive function [2]. Nelson-Siegel Model, on the other hand, provides a globally smooth curve and avoids overfitting.

In addition, the Nelson-Siegel model is a parametric model and only has 4 parameters: β0, β1, β2 control the level, slope, and curvature, and τ controls the exponential decay, which makes it computationally efficient. Cubic-Spline model requires more computation and can be less stable for sparse or noisy data as it ensures the internal connection of noise data [3].

The Nelson-Siegel model is also easier to interpret with its 4 parameters. For example, for the above curve given by the Nelson-Siegel model, we can see its level is approximately 4.82, slope is approximately 0.92, curvature is approximately -2.39 and decay is approximately 2.11. Unlike Nelson-Siegel, the parameters of a cubic spline have no intuitive economic meaning.

```
from nelson_siegel_svensson.calibrate import calibrate_ns_ols
```

```
t = np.array([0.5,1,3,5,10,20,30])
y = np.array(yields.loc["2024-05-03"])
```

```
curve, status = calibrate_ns_ols(t, y, tau0=1.0)
assert status.success
y_est = curve
t_est = np.linspace(0.5,30,100)
plt.plot(t, y, 'ro')
plt.plot(t_est, y_est(t_est))
plt.xlabel("Maturity")
plt.ylabel("Yield")
plt.title("NS Model Result")
curve
```

```
NelsonSiegelCurve(beta0=4.820017888453549, beta1=0.9216139083815179, beta2=-2.389113874880242, tau=2.1135354375132294)
```

g. Smoothing the yield curve using the Nelson-Siegel model is ethical when applied appropriately. The NSS model is widely adopted in financial markets and academic research for its ability to accurately represent the complex dynamics of interest rates over different maturity periods [1]. It is designed to create a smooth representation of the yield curve for better understanding. The purpose of smoothing is to reduce noise in the data caused by short-term market fluctuations and provide a clearer picture of the general structure of interest rates. However, it might become unethical if it is not used properly. For example, smoothing the yield curve to mask irregularities or anomalies during a financial crisis could be unethical if it is done to misinform investors or regulators. Also, ignoring or hiding deviations between actual yields and the smoothed curve on purpose is also unethical. If the model is applied in contexts where it is known to produce inaccurate results such as using it for highly irregular yield curves can also lead to unethical outcomes. Hence, if the Nelson-Siegel model is used appropriately and its limitations and assumptions are disclosed properly to stakeholders, it is ethical.

Task 3:

a.

```
[4]: mean = 0
     std_dev = 0.01
     num_variables = 5  # Number of random variables
     num_samples = 1  # Single sample of each

     np.random.seed(42)
     random_variables = np.random.normal(mean, std_dev, (num_samples, num_variables))

     print(random_variables)
```

```
[[ 0.00496714 -0.00138264  0.00647689  0.0152303  -0.00234153]]
```

5 uncorrelated Gaussian random variables are [0.00496714 -0.00138264 0.00647689 0.0152303 -0.00234153]

c.

The principal components are as follows:

PC1:[−0.2019,0.5291,−0.4179,−0.0877,0.7049]

PC2:[−0.3444,0.6311,0.4499,−0.3936,−0.3547]

PC3:[0.6942,0.0572,−0.1859,−0.6918,−0.0403]

PC4:[−0.2262,−0.4615,0.5139,−0.4343,0.5322]

PC5:[0.5545,0.3248,0.5695,0.4126,0.3040]

```
num_samples = 10  # Expand to 10 samples
random_variables_multiple = random_variables + np.random.normal(0, 0.0001, (num_samples, 5))
cov_matrix = np.cov(random_variables_multiple, rowvar=False)
pca = PCA()
pca.fit(cov_matrix)
principal_components = pca.components_
explained_variance = pca.explained_variance_ratio_

print("\nPrincipal Components (Eigenvectors):")
print(principal_components)
print("\nExplained Variance Ratios:")
print(explained_variance)
```

```
Principal Components (Eigenvectors):
[[-0.20194479  0.52911366 -0.41790326 -0.08770455  0.70492681]
 [-0.34440756  0.63113918  0.44985046 -0.39355854 -0.3546731 ]
 [ 0.69421065  0.05722716 -0.18585788 -0.6918283  -0.0403369 ]
 [-0.22622202 -0.4614553   0.51389309 -0.43425683  0.53218183]
 [ 0.55452403  0.32478627  0.56952802  0.41257023  0.30404048]]

Explained Variance Ratios:
[7.46026249e-01 1.51998062e-01 8.93912327e-02 1.25844556e-02
 2.37123461e-33]
```

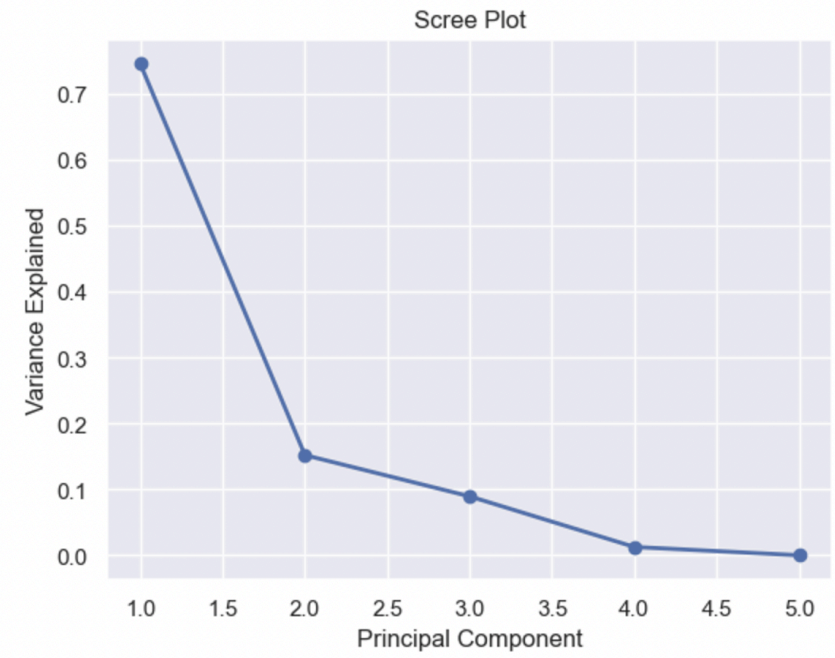
c.

The first principal component (PC1) explains 74.60% of the variance. The second principal component (PC2) explains 15.20% of the variance. The third principal component (PC3) explains 8.94% of the variance. The fourth principal component (PC4) explains 1.26% of the

variance. The fifth principal component (PC5) explains 2.37*10^(-31)% of the variance. We can see that the first three principal components explain more than 98% of the variance of and can be used to represent the data.

d.

```
[21]: PC_values = np.arange(pca.n_components_) + 1
      plt.plot(PC_values, pca.explained_variance_ratio_, 'o-', linewidth=2)
      plt.title('Scree Plot')
      plt.xlabel('Principal Component')
      plt.ylabel('Variance Explained')
      plt.show()
```



e, f.

We picked the United States Treasury securities (6 months, 1 year, 3 year, 5 year and 10 years) we used in Task 2

[22]:
```
yields = yields[["6 Month", "1 Year", "3 Year", "5 Year", "10 Year"]]
```

[23]:
```
yields = yields.dropna()
daily_yields = yields.pct_change()
daily_yields_filtered = daily_yields[daily_yields.index >= '2023-12-03']
daily_yields_filtered
```

[23]:

	6 Month	1 Year	3 Year	5 Year	10 Year
2023-12-04	0.015009	0.009901	0.020882	0.021739	0.014218
2023-12-05	-0.007394	-0.007843	-0.015909	-0.021277	-0.023364
2023-12-06	0.001862	0.001976	0.000000	-0.004831	-0.014354
2023-12-07	-0.003717	-0.003945	-0.004619	-0.002427	0.004854
2023-12-08	0.005597	0.015842	0.032483	0.031630	0.021739
...	...	...	...	...	...
2024-04-29	0.005556	-0.001919	-0.008264	-0.006410	-0.008565
2024-04-30	0.001842	0.009615	0.014583	0.015054	0.012959
2024-05-01	-0.001838	-0.007619	-0.016427	-0.016949	-0.012793
2024-05-02	-0.001842	-0.009597	-0.016701	-0.015086	-0.010799
2024-05-03	-0.001845	-0.007752	-0.016985	-0.019694	-0.017467

g.

[24]:
```
covariance_matrix = daily_yields_filtered.cov()
covariance_matrix
pca = PCA()
pca.fit(covariance_matrix)
principal_components = pca.components_
explained_variance = pca.explained_variance_ratio_

print("\nPrincipal Components (Eigenvectors):")
print(principal_components)
print("\nExplained Variance Ratios:")
print(explained_variance)
```

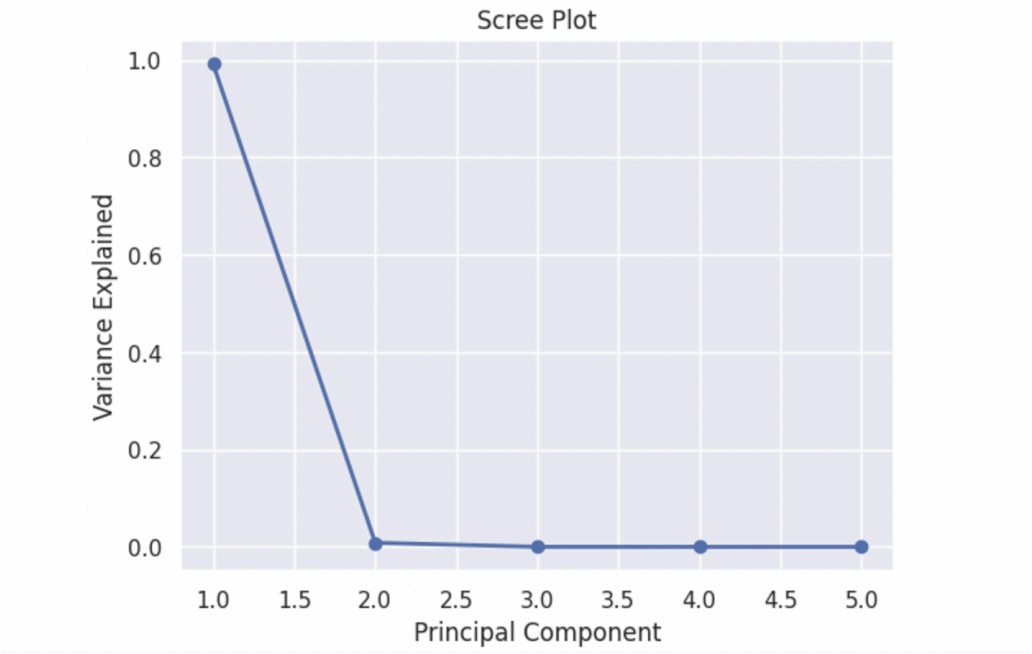
```
Principal Components (Eigenvectors):
[[-0.07930553 -0.26028494 -0.55415933 -0.59211031 -0.5179529 ]
 [-0.20249222 -0.57657443 -0.39882405  0.05788     0.68128417]
 [-0.48530221  0.72552298 -0.43448251 -0.03861165  0.21870586]
 [ 0.08136083 -0.07087925 -0.46972889  0.79048864 -0.37794051]
 [ 0.84295297  0.26154582 -0.35274146 -0.14032871  0.2773182 ]]

Explained Variance Ratios:
[9.91303253e-01 8.47130057e-03 1.55187392e-04 7.02589226e-05
 7.68298739e-35]
```

h.

The first principal component (PC1) explains 99.13% of the variance. The second principal component (PC2) explains 0.85% of the variance. The third principal component (PC3) explains 0.015% of the variance. The fourth principal component (PC4) explains 0.007% of the variance. The fifth principal component (PC5) explains 7.68 * 10^(-35) of the variance.

i.

```
[27]: PC_values = np.arange(pca.n_components_) + 1
      plt.plot(PC_values, pca.explained_variance_ratio_, 'o-', linewidth=2)
      plt.title('Scree Plot')
      plt.xlabel('Principal Component')
      plt.ylabel('Variance Explained')
      plt.show()
```



j.

For uncorrelated gaussian data, the first few components explain a relatively large portion of the variance, but the explained variance diminishes. For government securities data, the first principal component explains the majority of the variance (99.13%) and subsequent components contribute very little and the screeplot shows a clear "elbow" after the first component, indicating that most of the variance is captured by the first component. This is due to the fact that government yield data is highly structured with strong correlations between securities of different maturities because of economic factors. For uncorrelated gaussian data, principal components are less meaningful since the dataset lacks real-world structure or correlation.

Task 4:

a, b.

We pick the information technology sector (XLK) that tracks the performance of companies from the information technology sector within the S&P 500 Index. We find the 30 largest holdings and get the last 6 months of data.

[5]:
```
# The following is for Task 4
tickers = [
    "AAPL", "MSFT", "NVDA", "AVGO", "ORCL", "TSM", "ADBE", "CSCO", "CRM", "INTC",
    "TXN", "QCOM", "ACN", "AMD", "IBM", "AMAT", "NOW", "ADI", "NXPI", "LRCX",
    "PAYC", "SNPS", "FTNT", "PANW", "CDNS", "ANSS", "KEYS", "MU", "MPWR", "KLAC"
]
end_date = pd.Timestamp.today().strftime('%Y-%m-%d')
start_date = (pd.Timestamp.today() - pd.DateOffset(months=6)).strftime('%Y-%m-%d')
data = yf.download(tickers, start=start_date, end=end_date, interval="1d", group_by='ticker')
closing_prices = {ticker: data[ticker]['Close'] for ticker in tickers}
closing_prices_df = pd.DataFrame(closing_prices)
closing_prices_df
```

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

[5]:

Date	AAPL	MSFT	NVDA	AVGO	ORCL	TSM	ADBE	CSCO	CRM	INTC	...	PAYC	SNPS
2024-07-08	227.306534	465.401154	128.180191	173.548660	144.297211	185.355087	575.400024	45.401287	256.274780	33.775742	...	140.027573	615.48999C
2024-07-09	228.164581	458.713226	131.359726	172.301117	139.969177	183.259521	566.020020	45.145000	251.747070	34.371960	...	139.837997	607.940002
2024-07-10	232.454895	465.411133	134.889175	173.432343	141.352173	189.744904	564.549988	45.608284	251.906647	34.650192	...	139.957718	619.559998
2024-07-11	227.057098	453.881927	127.380325	169.581375	142.439713	183.239655	557.630005	46.091274	250.440613	33.288830	...	144.148071	608.73999C
2024-07-12	230.020386	452.733978	129.220047	169.056519	144.445206	186.070190	559.049988	46.702408	253.282913	34.272591	...	148.088989	614.049988
...	...	...	...	...	...	...	...	...	...	...	...	...	..
2024-12-30	252.199997	424.829987	137.490005	235.580002	166.910004	200.389999	445.799988	58.789391	335.739990	19.820000	...	204.440002	486.73999C
2024-12-31	250.419998	421.500000	134.289993	231.839996	166.639999	197.490005	444.679993	58.799324	334.329987	20.049999	...	204.970001	485.359985
2025-01-02	243.850006	418.579987	138.309998	231.979996	166.029999	201.580002	441.000000	58.700001	330.660004	20.219999	...	201.399994	482.75000C
2025-01-03	243.360001	423.350006	144.470001	232.550003	166.320007	208.610001	430.570007	58.860001	332.899994	20.559999	...	203.860001	493.70999

c.

To compute the daily returns.

```
[20]: daily_returns = closing_prices_df.pct_change().dropna()
      daily_returns
```

[20]:

Date	AAPL	MSFT	NVDA	AVGO	ORCL	TSM	ADBE	CSCO	CRM	INTC	...	PAYC	SNPS	FTNT	
2024-07-09	0.003775	-0.014370	0.024805	-0.007188	-0.029994	-0.011306	-0.016302	-0.005645	-0.017667	0.017652	...	-0.001354	-0.012267	0.002828	-0
2024-07-10	0.018804	0.014602	0.026869	0.006565	0.009881	0.035389	-0.002597	0.010262	0.000634	0.008095	...	0.000856	0.019114	-0.011447	0
2024-07-11	-0.023221	-0.024772	-0.055667	-0.022204	0.007694	-0.034284	-0.012258	0.010590	-0.005820	-0.039289	...	0.029940	-0.017464	-0.013425	-0
2024-07-12	0.013051	-0.002529	0.014443	-0.003095	0.014080	0.015447	0.002546	0.013259	0.011349	0.029552	...	0.027339	0.008723	0.012927	0
2024-07-15	0.016743	0.000904	-0.006190	0.007956	-0.011743	-0.011369	0.011913	0.000422	-0.004371	-0.000870	...	0.051742	0.003192	0.005877	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2024-12-30	-0.013263	-0.013240	0.003503	-0.025522	-0.012133	-0.006150	-0.001523	-0.007046	-0.008007	-0.023645	...	-0.011029	-0.008494	-0.009888	-0
2024-12-31	-0.007058	-0.007838	-0.023275	-0.015876	-0.001618	-0.014472	-0.002512	0.000169	-0.004200	0.011604	...	0.002592	-0.002835	-0.006833	-
2025-01-02	-0.026236	-0.006928	0.029935	0.000604	-0.003661	0.020710	-0.008276	-0.001689	-0.010977	0.008479	...	-0.017417	-0.005377	0.002858	-0
2025-01-03	-0.002009	0.011396	0.044538	0.002457	0.001747	0.034874	-0.023651	0.002726	0.006774	0.016815	...	0.012215	0.022703	0.028707	0
2025-01-06	0.006739	0.010630	0.034332	0.016599	-0.003788	0.054647	0.001417	-0.001529	-0.007119	-0.033560	...	0.015550	0.014887	0.004822	-0

126 rows × 30 columns

d.

To compute the covariance of returns

```
[21]: daily_returns_means = daily_returns.mean()
      daily_returns_stds = daily_returns.std()
      standardized_returns = (daily_returns - daily_returns_means) / daily_returns_stds

      standardized_returns_dvd_sqrt_n=(standardized_returns/math.sqrt(len(standardized_returns)-1))
      standardized_returns_cov = standardized_returns_dvd_sqrt_n.T@standardized_returns_dvd_sqrt_n
      standardized_returns_cov
```

[21]:

	AAPL	MSFT	NVDA	AVGO	ORCL	TSM	ADBE	CSCO	CRM	INTC	...	PAYC	SNPS	FTNT	PANW
AAPL	1.000000	0.560328	0.398547	0.367579	0.243876	0.359072	0.324499	0.246310	0.282489	0.229316	...	0.126635	0.411668	0.192342	0.402794
MSFT	0.560328	1.000000	0.496958	0.439703	0.540636	0.418683	0.390648	0.355489	0.386166	0.430174	...	-0.024213	0.558834	0.143533	0.502343
NVDA	0.398547	0.496958	1.000000	0.551871	0.427426	0.702929	0.329807	0.296197	0.412004	0.426441	...	0.047402	0.611778	0.013953	0.405113
AVGO	0.367579	0.439703	0.551871	1.000000	0.379655	0.646363	0.249623	0.223837	0.309744	0.355079	...	0.093969	0.525543	-0.011697	0.376971
ORCL	0.243876	0.540636	0.427426	0.379655	1.000000	0.300212	0.352013	0.362673	0.454833	0.379559	...	0.102359	0.513622	0.152683	0.466895
TSM	0.359072	0.418683	0.702929	0.646363	0.300212	1.000000	0.204235	0.186361	0.320130	0.403216	...	0.071797	0.536392	0.026247	0.300622
ADBE	0.324499	0.390648	0.329807	0.249623	0.352013	0.204235	1.000000	0.303947	0.422898	0.206875	...	0.210988	0.359240	0.143837	0.386327
CSCO	0.246310	0.355489	0.296197	0.223837	0.362673	0.186361	0.303947	1.000000	0.419540	0.431119	...	0.178611	0.299977	0.172520	0.340582
CRM	0.282489	0.386166	0.412004	0.309744	0.454833	0.320130	0.422898	0.419540	1.000000	0.327610	...	0.211285	0.528613	0.332525	0.500269
INTC	0.229316	0.430174	0.426441	0.355079	0.379559	0.403216	0.206875	0.431119	0.327610	1.000000	...	0.156316	0.562056	0.042875	0.371659
TXN	0.333014	0.418606	0.494471	0.401914	0.293677	0.493971	0.256041	0.429469	0.336473	0.575031	...	0.175050	0.550183	0.049052	0.289227
QCOM	0.491205	0.519853	0.712643	0.612914	0.353227	0.657477	0.285559	0.227439	0.351392	0.499657	...	0.142940	0.630778	0.033091	0.361432
ACN	0.214552	0.224685	0.179476	0.123968	0.349084	0.095089	0.288978	0.368876	0.284503	0.238643	...	0.213293	0.230371	0.171632	0.356737
AMD	0.406846	0.438485	0.631988	0.511143	0.358100	0.613231	0.188798	0.264168	0.408185	0.409596	...	0.065241	0.580890	0.061125	0.362177
IBM	0.194722	0.226739	0.242450	0.208581	0.331137	0.240670	0.203223	0.447895	0.353852	0.273097	...	0.347609	0.255100	0.188379	0.403585
AMAT	0.433198	0.486339	0.692181	0.579698	0.338287	0.707713	0.300493	0.210886	0.383517	0.556439	...	0.165194	0.749623	0.048856	0.341013
NOW	0.309432	0.360331	0.358643	0.282424	0.412972	0.299388	0.401788	0.278477	0.599416	0.277928	...	0.281817	0.420976	0.152259	0.480220
ADI	0.455914	0.524606	0.608762	0.505576	0.349738	0.603887	0.326727	0.431583	0.379976	0.615037	...	0.154899	0.656798	0.035436	0.327583
NXPI	0.419637	0.458670	0.565787	0.492073	0.246930	0.556857	0.256912	0.330330	0.311409	0.556686	...	0.081925	0.619672	-0.020880	0.234763
LRCX	0.385135	0.482235	0.650199	0.605076	0.307810	0.683399	0.306996	0.172852	0.356367	0.549879	...	0.124218	0.711504	0.027303	0.330308

e.

To compute PCA

```
[13]: pca = PCA()
pca.fit(standardized_returns_cov)

principal_components = pca.components_
explained_variance = pca.explained_variance_ratio_

print("\nPrincipal Components (Eigenvectors):")
print(principal_components)

print("\nExplained Variance Ratios:")
print(explained_variance)
```

```
Principal Components (Eigenvectors):
[[-7.54532350e-02 -1.09479628e-01 -2.09009390e-01 -1.88298810e-01
  -1.34183275e-02 -2.31012212e-01  1.70613508e-02  3.78098113e-02
   7.40922630e-03 -1.35096783e-01 -2.15093951e-01 -2.63182962e-01
   1.08486559e-01 -2.25816523e-01  8.74494630e-02 -2.79084789e-01
   2.36829663e-02 -2.44013101e-01 -2.71488208e-01 -2.95084329e-01
   9.59831545e-02 -1.96404025e-01  1.39996509e-01  3.35002295e-02
  -2.02994753e-01 -1.60585335e-01 -1.07922170e-01 -2.14961349e-01
  -2.67097925e-01 -2.83184623e-01]
 [-9.92030568e-02 -2.90914767e-01 -1.09056452e-01 -4.80525395e-02
  -3.66891556e-01  2.62997992e-02 -2.57301817e-01 -3.66377141e-02
  -3.32049374e-01 -5.69824669e-03  1.72724978e-01  6.41033336e-02
  -7.45289955e-02  2.65052320e-02 -7.24724337e-03  3.65018367e-02
  -3.45521852e-01  1.19866398e-01  1.59042314e-01  4.85706001e-02
   1.23405922e-01 -2.82808956e-01 -2.85969921e-02 -3.23669842e-01
  -2.90233239e-01 -2.81643755e-01  1.34970626e-02  7.40838951e-02
   4.86493214e-02  5.30122914e-02]
```

```
Explained Variance Ratios:
[6.61561661e-01 7.88690210e-02 5.73069803e-02 3.74154448e-02
 2.66998151e-02 2.40502321e-02 1.94563473e-02 1.59664947e-02
 1.31386968e-02 1.02402552e-02 9.11738029e-03 7.75253384e-03
 6.77863479e-03 6.23984044e-03 5.07614808e-03 4.46558192e-03
 3.33196235e-03 2.59584192e-03 2.40459967e-03 2.29067149e-03
 1.89966378e-03 1.00084217e-03 8.62959689e-04 5.70130150e-04
 4.16751353e-04 2.39598500e-04 1.65326675e-04 6.56532645e-05
 2.09316831e-05 2.58280607e-32]
```

f.

To compute SVD

```
[23]: U_st_return, s_st_return, VT_st_return = np.linalg.svd(standardized_returns_dvd_sqrt_n)
print("U matrix:")
print(U_st_return)
print("\nSingular values (S):")
print(s_st_return)
print("\nVt matrix:")
print(VT_st_return.T)
```

```
U matrix:
[[ 0.02770263  0.10654319 -0.02116893 ...  0.13669214  0.06796821
   0.0305015 ]
 [-0.07870497  0.05296426 -0.00977978 ... -0.02624099 -0.09543401
  -0.30576   ]
 [ 0.10474048 -0.09546045 -0.12562549 ...  0.14859127  0.05186406
   0.0651288 ]
 ...
 [ 0.0082343   0.08497417 -0.00656594 ...  0.89317108 -0.03719237
  -0.06885962]
 [-0.09441086  0.02404942 -0.03429764 ... -0.02847863  0.88706707
  -0.0469891 ]
 [-0.07674075  0.13161157  0.02574944 ... -0.05277643 -0.0222516
   0.69251012]]

Singular values (S):
[3.71995943 1.70637896 1.2306415  1.07043207 1.01858079 0.98274151
 0.95724865 0.87404264 0.86504717 0.79301445 0.77400563 0.73227637
```

Complete U matrix and Vt matrix can be found in PDF document with the output from the Jupyter notebook**.**

Analysis:

Returns in finance represent the percentage change in the price of an asset for a given period. Returns are essential because they indicate the profitability or loss of an investment and provide investors with a quantifiable measure of financial performance. In addition, returns also provide the basis for further analysis, such as risk assessment, portfolio optimization, and performance evaluation, hence it is important to modern finance.

Principal Component Analysis (PCA) and Singular Value Decomposition (SVD) are two important techniques in financial engineering. They both decompose a matrix into several components. PCA is primarily used for dimensionality reduction and its goal is to reduce the number of variables in a dataset while retaining original variance as much as possible. PCA identifies the key factors (principal components) that are orthogonal and explain most of the variation in the data, which simplifies data, reduces the noise and extracts key features. In financial engineering, it is very useful when we need to handle a large number of assets, since it allows us to identify patterns and relationships between stocks without being overwhelmed by the complexity of the data.The eigenvectors and eigenvalues in PCA provide insight about the data's structure. Eigenvectors show the directions of maximum variance in the dataset, and the corresponding eigenvalues indicate the magnitude of the variance along each eigenvector. For financial returns, the eigenvectors (principal components) represent the uncorrelated factors such as market-wide factors which drive movements of the stocks. The eigenvalues reflect how much each principal component contributes to explaining the overall variance in the data. For example, a large eigenvalue corresponds to a principal component that explains a significant portion of the variability in stock returns. As can be seen from the result in task e, the first principal component explains about 66.16% of the variability in stock returns and the second principal component explains about 7.89% of the variability in stock returns.

SVD is a more general matrix decomposition technique. It decomposes any matrix into three other matrices: U (left singular vectors), S (singular values), and Vt (right singular vectors). The relationship between these matrices provides important insight into the structure of the original data matrix. SVD helps identify underlying factors that explain the relationships between different stocks. The singular values represent the strength of the underlying factors (comparable to the eigenvalues in PCA), while the matrices U and Vt provide the directions of the data in the original space and the transformed space. The U matrix represents the left singular vectors, which can be seen as the "loadings" or importance of each stock in each factor. The Vt matrix represents the right singular vectors, which can be interpreted as the contributions of the principal components to each stock.

Both PCA and SVD help reduce the complexity of high-dimensional financial data, but they do so in different ways. PCA is specifically designed to find principal components that maximize

variance, whereas SVD decomposes the data matrix into factors that are not necessarily orthogonal but are still meaningful for understanding relationships in the data. They allow analysts to discover hidden patterns, identify key risk factors, and simplify the decision-making process, finally enabling more effective risk management and investment strategies. By understanding the underlying factors driving asset returns, investors can make more informed decisions and potentially improve their portfolio performance.

There is also one important connection between PCA and SVD [4]. If we let matrix B be standardized_returns_matrix / sqrt(n-1). Then the covariance of the standardized returns can be written as Bt*B. If we apply SVD on matrix B and let B = UΣVt, then we can compute Bt*BV = VΣ^2. This means the column vectors in V are the eigenvectors of Bt*B and the squared values in matrix Σ are eigenvalues of Bt*B. These results are also well supported by our data. In the code, standardized_returns_dvd_sqrt_n = B and standardized_returns_cov = Bt*B. The singular values we got in task f and the eigenvalues of Bt*B are respectively as follows:

```
[23]: U_st_return, s_st_return, VT_st_return = np.linalg.svd(standardized_returns_dvd_sqrt_n)
      print("U matrix:")
      print(U_st_return)
      print("\nSingular values (S):")
      print(s_st_return)
      print("\nVt matrix:")
      print(VT_st_return.T)
```

```
 [ 0.0082343   0.08497417 -0.00656594 ...  0.89317108 -0.03719237
  -0.06885962]
 [-0.09441086  0.02404942 -0.03429764 ... -0.02847863  0.88706707
  -0.0469891 ]
 [-0.07674075  0.13161157  0.02574944 ... -0.05277643 -0.0222516
   0.69251012]]

Singular values (S):
[3.71995943 1.70637896 1.2306415  1.07043207 1.01858079 0.98274151
 0.95724865 0.87404264 0.86504717 0.79301445 0.77400563 0.73227637
 0.70558309 0.69589064 0.65730011 0.64015118 0.61314037 0.56105951
 0.54828291 0.53275577 0.51894334 0.47170411 0.43122274 0.39580107
 0.36958009 0.34385779 0.27958136 0.23886386 0.21416184 0.15608039]

Vt matrix:
[[-1.48310310e-01 -7.49072060e-02  1.45795230e-01 -2.28376847e-01
   2.72802626e-01 -9.50397224e-02  5.44412145e-01 -2.45363524e-02
  -2.37922403e-01  2.70845810e-01 -2.69316177e-01 -1.52037533e-01
```

```
[18]: eigenvalues, eigenvectors = LA.eig(standardized_returns_cov)
      idx = np.argsort(eigenvalues)[::-1]
      eigenvalues = eigenvalues[idx]
      eigenvectors = eigenvectors[:, idx]
      eigenvalues
```

```
[18]: array([13.83809818,  2.91172917,  1.5144785 ,  1.14582481,  1.03750683,
             0.96578088,  0.91632498,  0.76395054,  0.7483066 ,  0.62887191,
             0.59908471,  0.53622869,  0.49784749,  0.48426379,  0.43204344,
             0.40979354,  0.37594111,  0.31478778,  0.30061415,  0.28382871,
             0.26930219,  0.22250477,  0.18595305,  0.15665849,  0.13658945,
             0.11823818,  0.07816574,  0.05705594,  0.04586529,  0.02436109])
```

It can be easily seen that the squared values in matrix Σ are indeed eigenvalues of Bt*B. For example, 3.71995943^2 = 13.83809818. We can also easily verify that the column vectors in V are the eigenvectors of Bt*B. We take the first two column vectors in V as an example and compare them with eigenvectors of Bt*B.

```
Vt matrix:
[[-1.48310310e-01 -7.49072060e-02  1.45795230e-01 -2.28376847e-01
   2.72802626e-01 -9.50397224e-02  5.44412145e-01 -2.45363524e-02
  -2.37922403e-01  2.70845810e-01 -2.69316177e-01 -1.52037533e-01
  -5.65399470e-02  2.17234924e-01  4.50495780e-02  7.48937144e-02
   1.77465507e-01 -8.22425862e-02 -6.69547395e-02 -2.74913936e-01
  -1.18427885e-02  1.61206409e-01  2.43954996e-01 -8.50909001e-02
  -7.48481499e-02 -1.16443136e-01  9.57334702e-03  5.89181637e-02
   4.99576244e-04 -3.12670405e-02]
 [-1.78257710e-01 -9.74435360e-02  2.34714659e-01 -3.67420905e-01
  -1.03606197e-01 -3.59333381e-02  1.64621336e-01 -1.39334693e-01
  -2.01596213e-01  1.18293484e-01  1.38647185e-01  6.08141184e-02
  -1.54166779e-01 -5.01732346e-03  1.06740701e-02 -3.39273229e-01
   1.40667380e-01  8.40684170e-02  2.96787901e-01  4.95399529e-01
   1.04474473e-01 -6.97464766e-02 -3.26800293e-01 -2.66879945e-02
   6.59788867e-02  2.26731092e-02  2.62812864e-02 -2.93006765e-02
  -5.21705506e-02 -2.72844140e-02]
 [-2.06758318e-01  5.91166283e-02  1.36396592e-01 -2.13106618e-02
```

[19]:
```
eigenvectors
```

[19]:
```
array([[-1.48310310e-01, -7.49072060e-02, -1.45795230e-01,
        -2.28376847e-01, -2.72802626e-01,  9.50397224e-02,
        -5.44412145e-01, -2.45363524e-02, -2.37922403e-01,
         2.70845810e-01, -2.69316177e-01, -1.52037533e-01,
        -5.65399470e-02, -2.17234924e-01, -4.50495780e-02,
         7.48937144e-02, -1.77465507e-01, -8.22425862e-02,
        -6.69547395e-02,  2.74913936e-01, -1.18427885e-02,
         1.61206409e-01,  2.43954996e-01, -8.50909001e-02,
         7.48481499e-02, -1.16443136e-01,  9.57334702e-03,
         5.89181637e-02, -4.99576244e-04, -3.12670405e-02],
       [-1.78257710e-01, -9.74435360e-02, -2.34714659e-01,
        -3.67420905e-01,  1.03606197e-01,  3.59333381e-02,
        -1.64621336e-01, -1.39334693e-01, -2.01596213e-01,
         1.18293484e-01,  1.38647185e-01,  6.08141184e-02,
        -1.54166779e-01,  5.01732346e-03, -1.06740701e-02,
        -3.39273229e-01, -1.40667380e-01,  8.40684170e-02,
         2.96787901e-01, -4.95399529e-01,  1.04474473e-01,
        -6.97464766e-02, -3.26800293e-01, -2.66879945e-02,
        -6.59788867e-02,  2.26731092e-02,  2.62812864e-02,
        -2.93006765e-02,  5.21705506e-02, -2.72844140e-02],
```

Works Cited

[1] Pape. "Understanding the Nelson-Siegel-Svensson (NSS) Model for Bond Yield Curve Analysis." Medium, 2024 May 12. https://medium.com/@pape14/understanding-the-nelson-siegel-svensson-nss-model-for-bond-yield-curve-analysis-2a23202cbf6b.

[2] Klein, William, "An Evaluation of Knot Placement Strategies for Spline Regression" (2021). CMC Senior Theses. 2545. https://scholarship.claremont.edu/cmc_theses/2545

[3] Hao, Wu & Deng, Rui & Song, Linghu & Ruixiang, Yang & Jinhai, Zhang & Juan, Cao. (2021). Data processing method of noise logging based on cubic spline interpolation. Applied Mathematics and Nonlinear Sciences. 6. 10.2478/amns.2021.1.00014.

[4] Brunton, Steven L., et al. "Chapter 1: Singular Value Decomposition (SVD) and Principal Components Analysis (PCA)." University of Washington, 2015, https://faculty.washington.edu/sbrunton/me565/pdf/CHAPTER1.pdf.