Problem statement: Developing representation learning model to extract meaningful features from financial securities data, enhancing informed decision-making in dynamic market environments

Loading libraries

In [1]:
```
# library to display a progress bar
import tqdm

# importing gensim to train a Word2Vec model
import gensim

# library for making HTTP requests
import requests

# library to operate large, multi-dimensional arrays and matrices
import numpy as np

# library for data manipulation and analysis
import pandas as pd

# library to access the financial data available on Yahoo Finance.
import yfinance as yf

# library for statistical data visualization
import seaborn as sns

# library to create, manipulate, and study complex networks.
import networkx as nx

# library for pulling data out of HTML and XML files
from bs4 import BeautifulSoup

# library for data visualization
import matplotlib.pyplot as plt

# importing word2vec algorithm from gensim
from gensim.models import Word2Vec
```

In [2]:
```
# library for machine learning on graphs and networks
from stellargraph import StellarGraph

# class used to perform biased random walks on the graph/networks
from stellargraph.data import BiasedRandomWalk
```

Fetching the list of stock tickers

In [3]:
```
# retrieving the content of the web page of a given URL.
resp = requests.get('http://en.wikipedia.org/wiki/List_of_S%26P_500

# using the BeautifulSoup library to parse the HTML content of the
soup = BeautifulSoup(resp.text, 'lxml')

# searching for a <table> element with the attribute class set to '
table = soup.find('table', {'class': 'wikitable sortable'})

# creating an empty list to store all the tickers mentioned in abov
tickers = []

# iterate over each row in the above table, excluding the header ro
for row in table.findAll('tr')[1:]:

    # extracting the text value from the first cell of each row in
    ticker = row.findAll('td')[0].text.strip('\n')

    # then appending the ticker into the empty list tickers
    tickers.append(ticker)

# replacing any occurrences of the dot character '.' with the hyphe
tickers = [ticker.replace('.', '-') for ticker in tickers]
```

Fetching the EOD price data for above tickers

In [4]:
```
# defining start date from when we want to fetch price data
start_date = '2022-01-01'

# defining end date from when we want to fetch price data
end_date = '2022-12-31'

# downloanding price data for all S&P 500 tickers for above time pe
data = yf.download(tickers, start=start_date, end=end_date)

# fetching only the EOD close prices
price_data = data['Close']
```

```
[*********************100%***********************]  503 of 503 com
pleted
```

In [5]:
```
# printing first 5 rows of the dataframe
price_data.head()
```

Out[5]:

Date	A	AAL	AAP	AAPL	ABBV	ABC	ABT
2022-01-03	156.479996	18.750000	236.779999	182.009995	135.419998	132.619995	139.039993
2022-01-04	151.190002	19.020000	237.050003	179.699997	135.160004	131.360001	135.770004
2022-01-05	148.600006	18.680000	236.449997	174.919998	135.869995	132.500000	135.160004
2022-01-06	149.119995	18.570000	241.649994	172.000000	135.229996	130.449997	135.139999
2022-01-07	145.149994	19.280001	238.089996	172.169998	134.880005	133.119995	135.559998

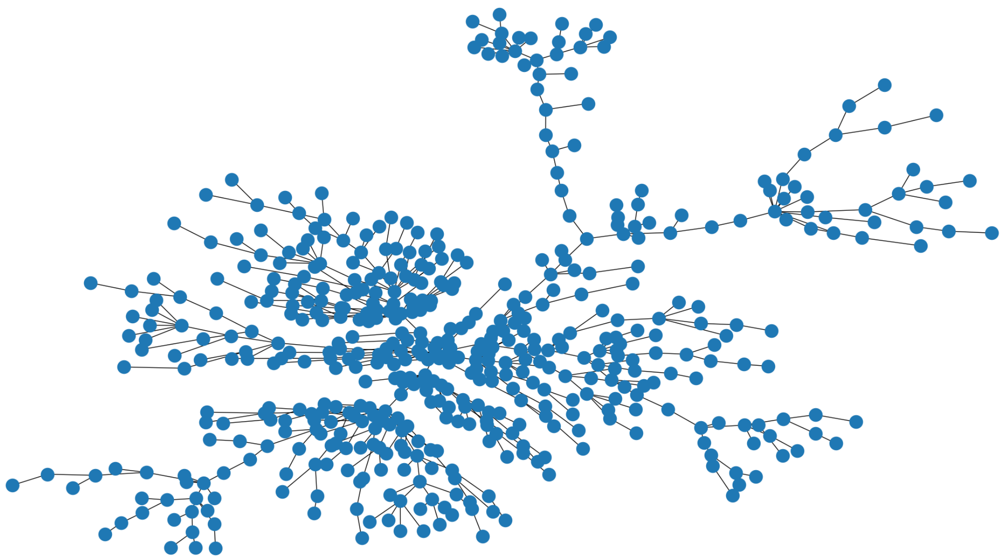
5 rows × 503 columns

Modeling the data

In [6]:
```
# computing the logarithmic returns of a financial time series data
log_returns_data = np.log(price_data / price_data.shift(1))

# computing the correlation matrix of the logarithmic returns data.
log_return_correlation = log_returns_data.corr()

# computing a distance matrix based on the correlation matrix of lo
distance_matrix = np.sqrt(2 * (1 - log_return_correlation))

# creating a graph object from the above distance matrix using the 
distance_graph = nx.Graph(distance_matrix)

# filtering the above fully connected graph using a minimum spannin
distance_graph_filtered = nx.minimum_spanning_tree(distance_graph)
```

Visualizing the graph

In [7]:
```
fig = plt.figure(figsize=(22, 12))
nx.draw_kamada_kawai(distance_graph_filtered)
```



Creating a dataframe of source and target nodes with correlation strength from above MST

In [8]:
```
# extracting the edges of the above graph and storing them in a list
edges = list(distance_graph_filtered.edges)

# creating an empty list to store correlation strength between two n
list_edges = []

# looping through each edge
for edge in edges:
    list_edges.append((edge[0], # storing first node of the edge
                       edge[1], # storing second node of the edge
                       log_return_correlation.loc[edge[0]][edge[1]]))

# creating a dataframe using the above list of edges
edges = pd.DataFrame(list_edges)

# renamning the columns
edges.columns = ['source', 'target', 'weight']
```

In [17]:
```
list_edges
```

Out[17]:
```
[('A', 'MTD', 0.8581709889119605),
 ('A', 'WAT', 0.8440641715920576),
 ('AAL', 'UAL', 0.9295164335957912),
 ('AAL', 'DAL', 0.925010492630063),
 ('AAL', 'NCLH', 0.7844245969677757),
 ('AAP', 'GPC', 0.6769399805551117),
 ('AAPL', 'MSFT', 0.8244061101136773),
 ('ABBV', 'GEHC', 0.5793919017928988),
 ('ABC', 'MCK', 0.8378683941978287),
 ('ABC', 'CAH', 0.7025876330164261),
 ('ABT', 'DHR', 0.7603544765861598),
 ('ABT', 'BDX', 0.6612509411265215),
 ('ABT', 'VRTX', 0.5007801708627065),
 ('ACGL', 'EG', 0.8219958901195606),
 ('ACGL', 'CB', 0.7674935248230313),
 ('ACN', 'APH', 0.813460696100857),
 ('ACN', 'PAYX', 0.7868162594076713),
 ('ACN', 'BR', 0.7693559084224082),
 ('ACN', 'CTSH', 0.7511336970019126),
```

Creating sentence like structures using random walk algorithm

In [9]:
```
# initializing a StellarGraph object with the specified edges to pe
G = StellarGraph(edges=edges)
```

In [10]:
```
# printing some basic information about the Graph
print(G.info())
```

```
StellarGraph: Undirected multigraph
 Nodes: 503, Edges: 502

 Node types:
  default: [503]
    Features: none
    Edge types: default-default->default

 Edge types:
    default-default->default: [502]
        Weights: range=[0.336561, 0.997779], mean=0.765147, std=0.
105559
        Features: none
```

Generating random walks

In [11]:
```
# creating an object to generate random walks
rw = BiasedRandomWalk(G)

# generating random walks on the graph G
weighted_walks = rw.run(
    nodes = G.nodes(), # specifying the starting nodes for the walk
    length=100, # the number of steps in each walk
    n=50, # the number of walks to start from each node
    p=0.5, # the return parameter, which controls the likelihood of
    q=2.0, # the in-out parameter, which allows the search to diffe
    weighted=True, # weights will be used when choosing the next st
    seed=42 # setting a seed for the random number generator, ensur
)
```

Training a Word2Vec model

In [12]:
```
weighted_model = Word2Vec(
    weighted_walks, # training data for the model
    window=5, # model will learn to predict a node based on the 5 n
    min_count=0, # don't ignore any nodes
    sg=1, # defines the training algorithm, which is skip-gram in t
    workers=8 # number of worker threads to train the model
)
```

Results

In [13]:
```
# finding out the nodes that are most similar to the node 'JPM'.
weighted_model.wv.most_similar('ABT')
```

Out[13]:
```
[('BDX', 0.9155576825141907),
 ('DHR', 0.9105576276779175),
 ('VRTX', 0.901513934135437),
 ('WST', 0.874254047870636),
 ('MTD', 0.8547109365463257),
 ('INCY', 0.850189745426178),
 ('REGN', 0.8457857370376587),
 ('TMO', 0.8113499879837036),
 ('RVTY', 0.7683820128440857),
 ('HOLX', 0.7674034833908081)]
```

Saving the embeddings

In [14]:
```
# saving the trained Word2Vec model
weighted_model.save('weighted_model')

# loading a previously saved Word2Vec model from a file named 'weig
model = gensim.models.word2vec.Word2Vec.load('weighted_model')

# opening the file 'embeddings.tsv' in write mode
with open('embeddings.tsv', 'w+') as tensors:

    # opening the file 'metadata.tsv' in write mode
    with open('metadata.tsv', 'w+') as metadata:

        # iterating over each word in the vocabulary of the Word2Ve
        for word in model.wv.index_to_key:

            # encodes the word as bytes
            encoded = word.encode()

            # writing each word to the 'metadata.tsv' file, followe
            metadata.write(word + '\n')

            # converting the word's embedding vector to a string re
            vector_raw = '\t'.join(map(str, model.wv[word]))

            # writing the string representation of the embedding ve
            tensors.write(vector_raw + '\n')
```

In [ ]: