

Final_submission

March 28, 2023

1 TOWER RESEARCH SOLUTION

1.0.1 Submitted by Himanshu Kumar

Github : <https://github.com/Vorlde/Tower-Research>

Drive : <https://tinyurl.com/3sss6zaw>

1.1 1. Explore Data

```
[ ]: #import libraries
import numpy as np
import pandas as pd
# import cvxpy as cp
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from scipy.stats import shapiro, skew, kurtosis
from scipy.cluster.hierarchy import linkage, dendrogram
from scipy.cluster.hierarchy import fcluster
from statsmodels.tsa.stattools import kpss
import seaborn as sns
import statsmodels.api as sm
import statsmodels.formula.api as smf

#settings
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
plt.rcParams["figure.figsize"] = (12, 9)
import warnings
warnings.filterwarnings('ignore')
```

Let's dig the data!

```
[ ]: df_index = pd.read_csv("data_challenge_index_prices.csv")
df_stock = pd.read_csv("data_challenge_stock_prices.csv")

#Compute returns and stndardize them
returns_stock = df_stock.pct_change().fillna(0)
```

```

returns_stock = ((returns_stock - returns_stock.mean()) / returns_stock.
↪std())*100
returns_index = df_index.pct_change().fillna(0)
returns_index = (((returns_index - returns_index.mean()) / returns_index.
↪std())*100)

```

```

[ ]: # returns_stock.info()
# returns_index.info()

```

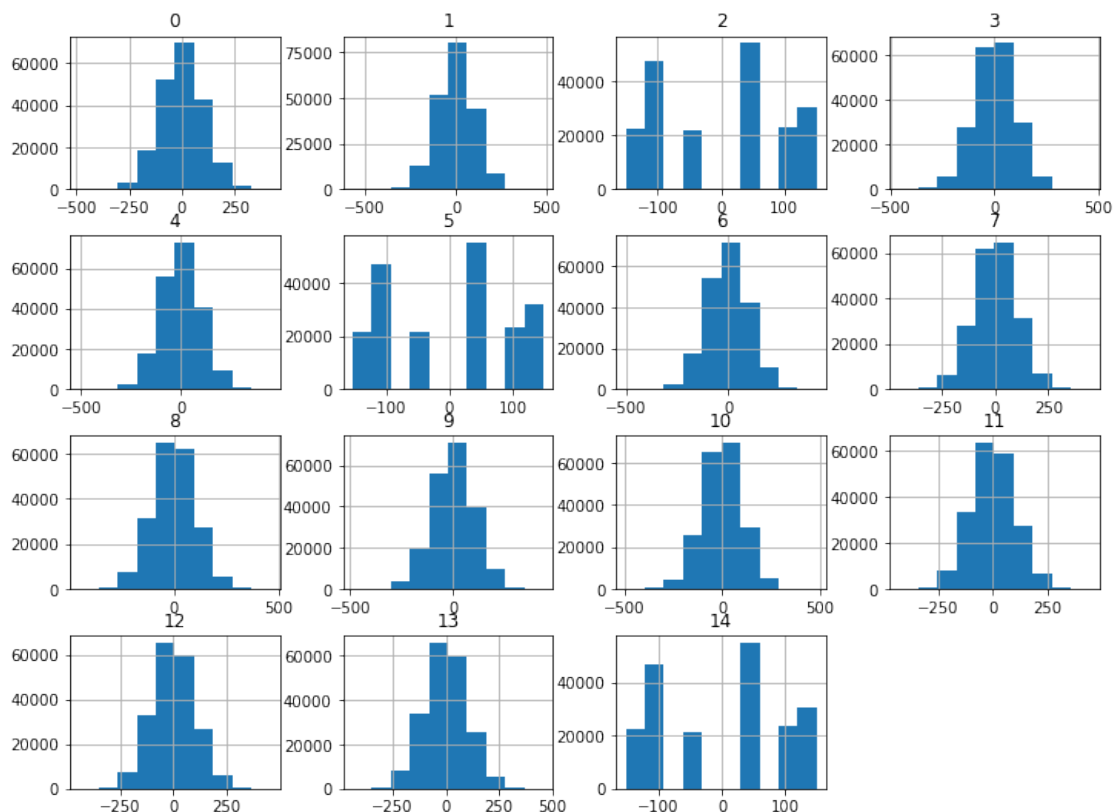
All non - null Values Data type set to float We have a clear visual confirmation that data doesn't have outliers as all stock price lie within 0-250 range that is permissible, so we are not going to perform statistical tests for outlier detection, So we have a clean and nice ready to go data, is it really nice? We need to check some more things wrt to time series

As we know in Finance we depend heavily on Normal distribution, so it's important to look at returns distribution to infer our data better

```

[ ]: index_corr = returns_index.corr()
returns_index.hist();

```



We have a culprits here lets catch them, we can see 2,5,14 are clearly a non normal distribution, lets call them **special indices** but we need to confirm this statistically and for that we will use

Shapiro-wilk as our normality test.

```
[ ]: # perform Shapiro-Wilk test for each stock and index

def normality_test(df,significance_level = 0.05):

    sw_results = df.apply(lambda x: shapiro(x))
    p_values = sw_results.apply(lambda x: x[1])

    # Get column indices where p-value is less than 0.05
    non_normal_columns = np.where(p_values < 0.05)[0]
    non_normal_columns = non_normal_columns.astype(int)

    return (non_normal_columns)
```

```
[ ]: print(normality_test(returns_stock))
      print(normality_test(returns_index))
```

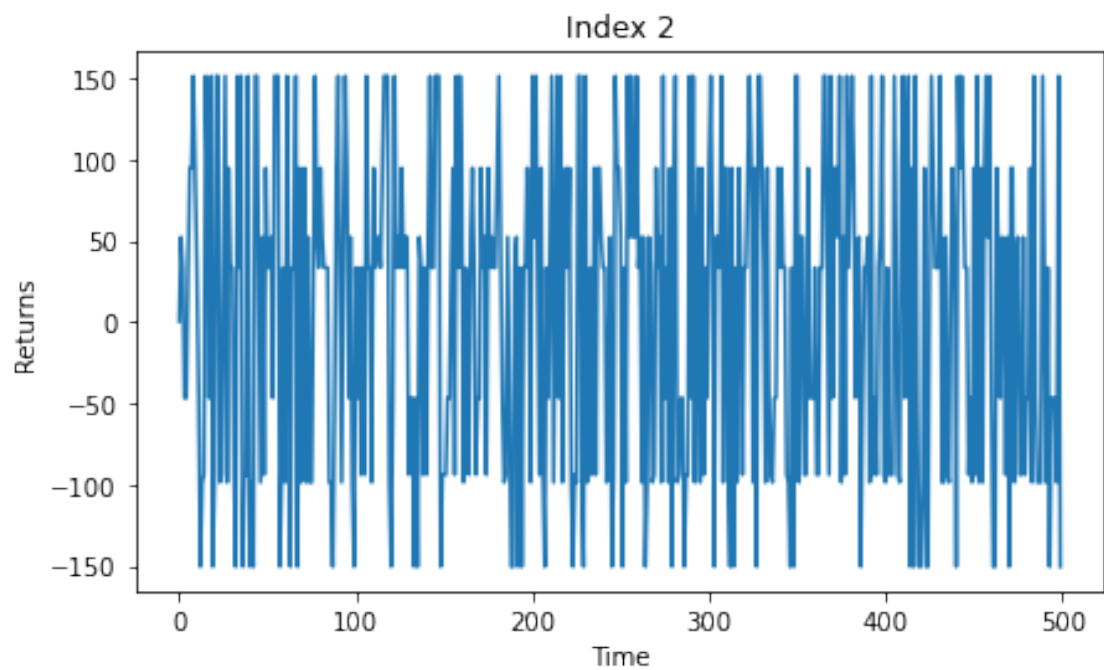
```
[]
[ 1  2  3  5  9 10 11 12 13 14]
```

We see p-values less than threshold hence we reject the null hypothesis that data comes from normal distribution. Inferences : Stocks are normally distributed, Some indices are not

```
[ ]: special_indexes = [2,5,14]

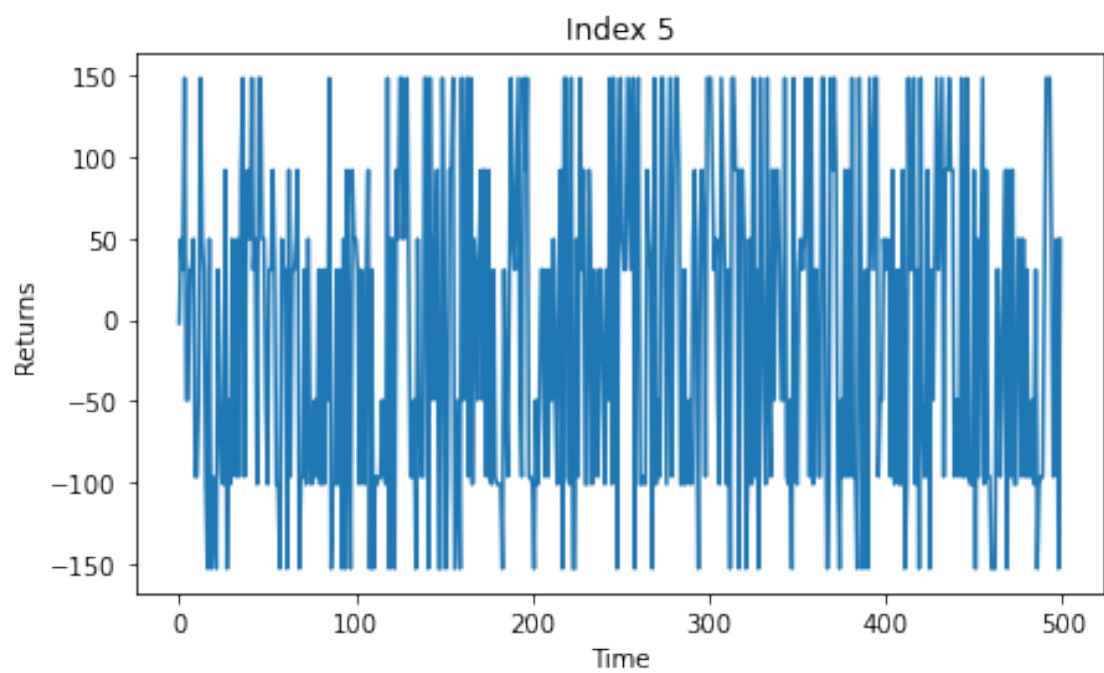
for i in special_indexes:
    data = returns_index.iloc[:,i]
    plt.rcParams["figure.figsize"] = (7, 4)
    fig, ax = plt.subplots()
    ax.plot(returns_index.iloc[:500,i])
    ax.set_xlabel("Time")
    ax.set_ylabel("Returns")
    ax.set_title(f"Index {i}")
    skewness = skew(data)
    kurt = kurtosis(data)
    print("Skewness: ", skewness)
    print("Kurtosis: ", kurt)
    plt.show()
```

```
Skewness:  0.03362579367076073
Kurtosis: -1.3149209268902733
```

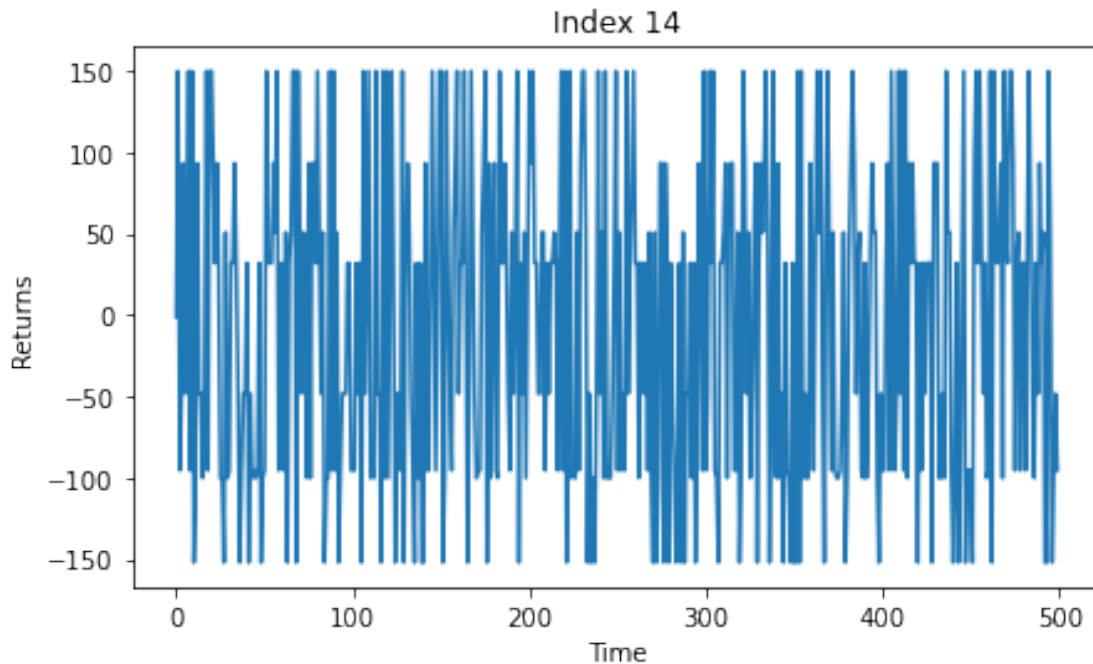


Skewness: 0.007279169710851204

Kurtosis: -1.3135858562573255



Skewness: 0.00903016153872343
Kurtosis: -1.3144583103743086



Nice !!! These are indeed special indices, what can we do with this information? Looks like returns are periodic in nature for these indices...How can we use this data to exploit, Let's Try to Decompose this via Fast Fourier Transform FFT

```
[ ]: def compute_dominant_freq(data):  
  
    # Apply the Fourier transform to the detrended data  
    fft = np.fft.fft(data)  
  
    # Compute the frequency axis  
    freq = np.fft.fftfreq(data.size)  
  
    # Find the index of the maximum amplitude in the frequency spectrum  
    max_idx = np.argmax(abs(fft))  
  
    # Get the dominant frequency from the frequency axis  
    dominant_freq = freq[max_idx]  
  
    return dominant_freq  
  
[ ]: dominant_freq = compute_dominant_freq(returns_index.iloc[:,2].values)  
    dominant_freq
```

```
[ ]: 0.008475
```

2 2. Compute M

Now After exploring We move to compute M, So Here are the things to try to get No. of Sectors Out of these 100 stocks to explain Indexes: 1. Decomposition of Correlation Matrix of returns 2. Clustering on the basis of Correlation Matrix of returns 3. Denoise and do the FFT of every stock and get the clusters of frequency (To try)

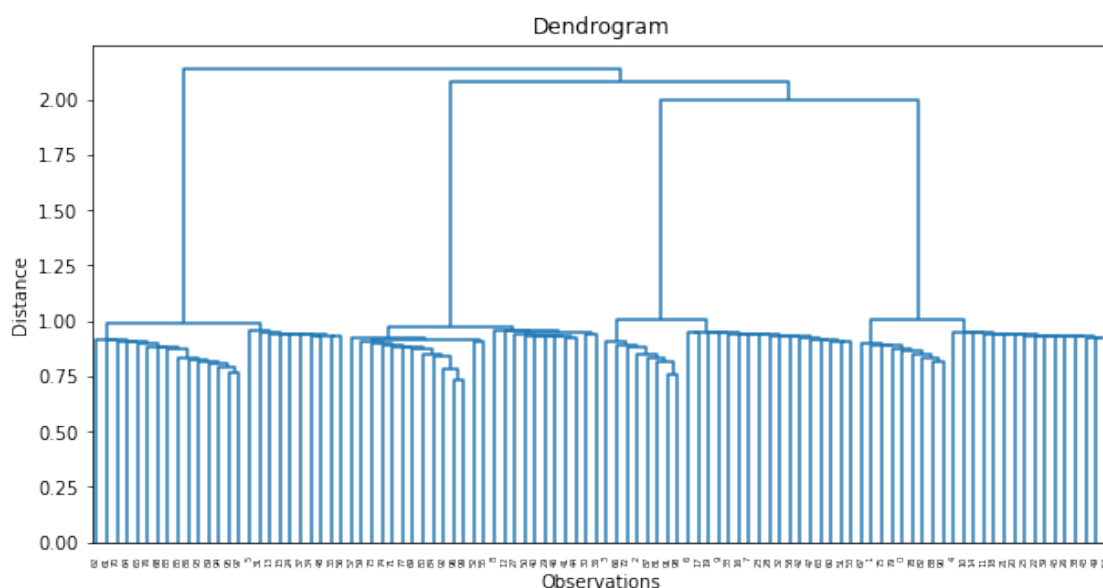
2.0.1 2.1 Clustering on the basis of Correlation Matrix of return

```
[ ]: # compute the correlation matrix
returns_corr = returns_stock.corr()

# Convert the correlation matrix to a distance matrix
dist_matrix = np.sqrt((1 - returns_corr.abs()).clip(0)) # Distance = sqrt(1 - |correlation|)
dist_array = dist_matrix.values[np.triu_indices_from(dist_matrix, k=1)]

# Perform hierarchical clustering
Z = linkage(dist_array, method='ward')

# Plot the dendrogram
plt.figure(figsize=(10, 5))
dendrogram(Z, color_threshold=0)
plt.title("Dendrogram")
plt.xlabel("Observations")
plt.ylabel("Distance")
plt.show()
```



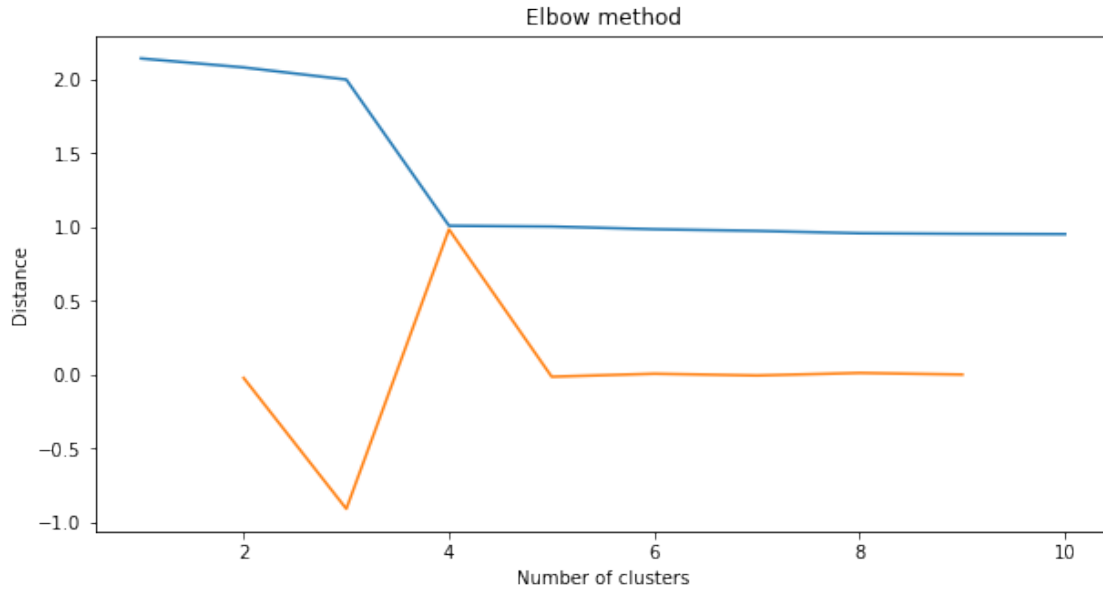
```
[ ]: # Determine the optimal number of clusters using the elbow method
last = Z[-10:, 2]
last_rev = last[::-1]
idxs = np.arange(1, len(last) + 1)
plt.figure(figsize=(10, 5))
plt.plot(idxs, last_rev)

# Compute the second derivative of the distances
acceleration = np.diff(last, 2)
acceleration_rev = acceleration[::-1]
plt.plot(idxs[:-2] + 1, acceleration_rev)
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Distance')
plt.show()

# Based on the elbow plot, choose the optimal number of clusters
k = acceleration_rev.argmax() + 2
print("Optimal number of clusters:", k)

# Perform agglomerative hierarchical clustering with k clusters
clusters = fcluster(Z, k, criterion='maxclust')
returns_corr['cluster'] = clusters

# Group the stocks by sector based on the cluster labels
sectors = returns_corr.groupby('cluster').groups
for sector, stocks in sectors.items():
    # Convert the stock indexes from string to integer
    stocks = [int(stock) for stock in stocks]
    print(f"Sector {sector}: {stocks}")
```



Optimal number of clusters: 4

Sector 1: [5, 13, 15, 24, 31, 34, 35, 37, 48, 56, 61, 62, 64, 65, 68, 70, 76, 83, 85, 86, 89, 93, 94, 95, 97]

Sector 2: [8, 12, 27, 29, 30, 36, 40, 41, 44, 46, 50, 52, 55, 57, 59, 69, 71, 73, 74, 77, 80, 84, 92, 96, 99]

Sector 3: [2, 3, 6, 7, 9, 16, 17, 19, 23, 28, 32, 33, 42, 47, 51, 53, 58, 60, 63, 66, 72, 81, 87, 91, 98]

Sector 4: [0, 1, 4, 10, 11, 14, 18, 20, 21, 22, 25, 26, 38, 39, 43, 45, 49, 54, 67, 75, 78, 79, 82, 88, 90]

2.0.2 2.2 Decomposition of Correlation Matrix of return

```
[ ]: eigenvalues, eigenvectors = np.linalg.eig(returns_corr)
M = np.sum(eigenvalues > 2)
print("Optimal number of clusters:", M)
```

Optimal number of clusters: 4

```
[ ]: # Optimal = 4, suseptibility to threshold
```

```
[ ]: kmeans = KMeans(n_clusters=4, random_state=0)
kmeans.fit(returns_corr)

# The predicted cluster labels for each stock
stock_clusters = kmeans.labels_
```



```
[ ]: # Map sector labels to stock names
sector_map = {}
for i in range(len(stock_clusters)):
    if stock_clusters[i] not in sector_map:
        sector_map[stock_clusters[i]] = []
        sector_map[stock_clusters[i]].append(i)

# Print out the mapping of sectors to stock names
for sector in sector_map:
    print("Sector {}: {}".format(sector, sector_map[sector]))
```

```
Sector 2: [0, 1, 4, 10, 11, 14, 18, 20, 21, 22, 25, 26, 38, 39, 43, 45, 49, 54,
67, 75, 78, 79, 82, 88, 90]
Sector 3: [2, 3, 6, 7, 9, 16, 17, 19, 23, 28, 32, 33, 42, 47, 51, 53, 58, 60,
63, 66, 72, 81, 87, 91, 98]
Sector 1: [5, 13, 15, 24, 31, 34, 35, 37, 48, 56, 61, 62, 64, 65, 68, 70, 76,
83, 85, 86, 89, 93, 94, 95, 97]
Sector 0: [8, 12, 27, 29, 30, 36, 40, 41, 44, 46, 50, 52, 55, 57, 59, 69, 71,
73, 74, 77, 80, 84, 92, 96, 99]
```

We see similar No. of clusters and the sectors being clustered from 2 methods and we can be sure about M to be 4, so we have 4 sectors with 25 stocks in each

2.0.3 2.3 M via FFT idea explore

To apply FFT we need to be sure our data is stationary and since we are applying it on returns which are stationary but for the sake of completeness let's check it statistically:

```
[ ]: # Define a function to perform the KPSS test on a single column of data

def kpss_test(data):
    # KPSS test
    kpss_result = kpss(data)
    return pd.Series({
        "KPSS Statistic": kpss_result[0],
        "p-value": kpss_result[1],
        "Lags Used": kpss_result[2],
        "Critical Values": kpss_result[3]
    })

# Apply the KPSS test to each column of the DataFrame
results = returns_stock.iloc[:,12:30].apply(kpss_test)
results
```

```
[ ]: KPSS Statistic      12 \
      p-value           0.065254
      Lags Used         0.1
                        29
```

Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	13 \
KPSS Statistic	0.117732	
p-value	0.1	
Lags Used	3	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	14 \
KPSS Statistic	0.459744	
p-value	0.051403	
Lags Used	25	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	15 \
KPSS Statistic	0.074597	
p-value	0.1	
Lags Used	4	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	16 \
KPSS Statistic	0.095594	
p-value	0.1	
Lags Used	19	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	17 \
KPSS Statistic	0.162247	
p-value	0.1	
Lags Used	9	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	18 \
KPSS Statistic	0.1284	
p-value	0.1	
Lags Used	2	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	19 \
KPSS Statistic	0.050397	
p-value	0.1	
Lags Used	14	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	20 \
KPSS Statistic	0.067251	
p-value	0.1	

Lags Used	26	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
KPSS Statistic	0.225199	21 \
p-value	0.1	
Lags Used	30	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
KPSS Statistic	0.075188	22 \
p-value	0.1	
Lags Used	11	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
KPSS Statistic	0.101921	23 \
p-value	0.1	
Lags Used	13	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
KPSS Statistic	0.644177	24 \
p-value	0.01862	
Lags Used	10	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
KPSS Statistic	0.475641	25 \
p-value	0.047153	
Lags Used	2	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
KPSS Statistic	0.023388	26 \
p-value	0.1	
Lags Used	31	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
KPSS Statistic	0.071768	27 \
p-value	0.1	
Lags Used	9	
Critical Values	{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.674}	
KPSS Statistic	0.320473	28 \

```

p-value                                0.1
Lags Used                              12
Critical Values  {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%...'

                                29
KPSS Statistic                        0.083337
p-value                                0.1
Lags Used                              14
Critical Values  {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%...'

```

The test is based on the null hypothesis that the time series is stationary, and the alternative hypothesis that the time series has a unit root and is non-stationary.

```

[ ]: data = returns_stock["0"] #Try for 1st stock

# Apply the Fourier transform
fft = np.fft.fft(data)

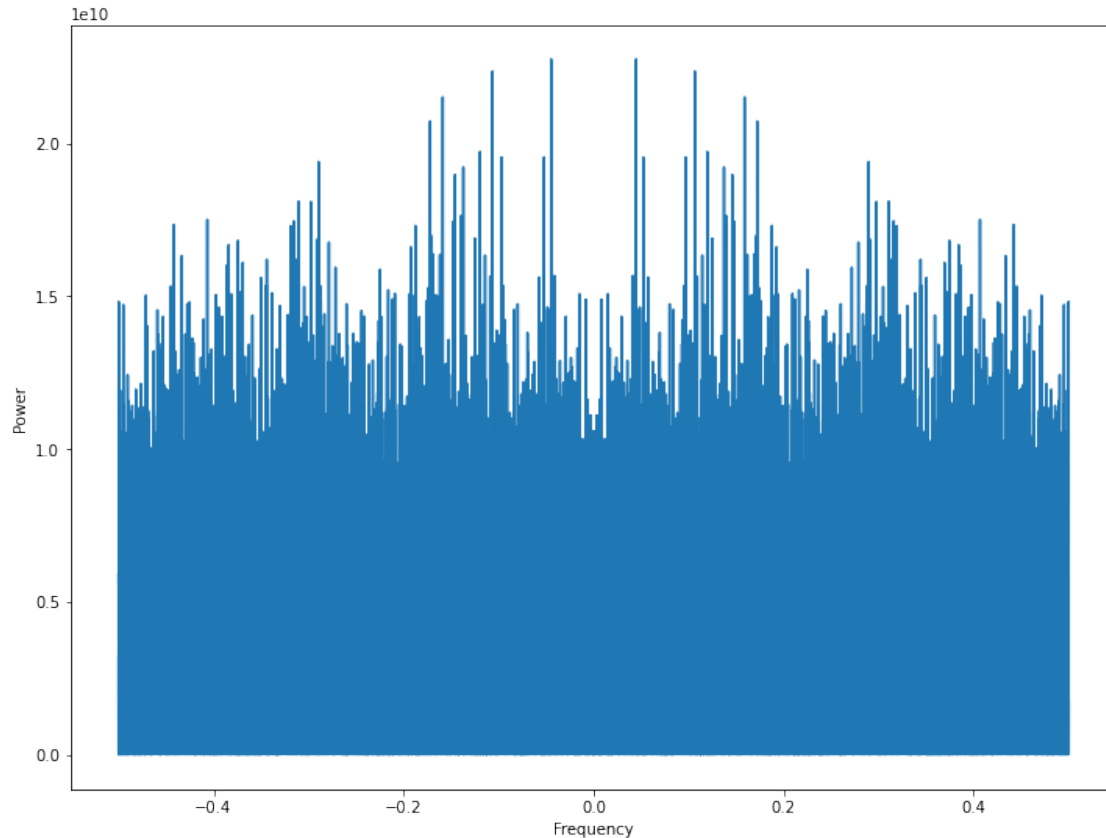
# Compute the frequency axis
freq = np.fft.fftfreq(data.size)

# Compute the power spectrum
power_spectrum = np.abs(fft) ** 2

# Plot the power spectrum against the frequency axis
plt.plot(freq, power_spectrum)
plt.xlabel('Frequency')
plt.ylabel('Power')
plt.show()

# Identify the dominant frequencies
dominant_freqs = freq[np.argsort(power_spectrum)[::-1][:5]] # get the 5 highest
↪ power frequencies
print('Dominant frequencies:', dominant_freqs)

```



Dominant frequencies: [-0.04474 0.04474 0.10635 -0.10635 0.15877]

We have captured 5 Dominant frequencies but We see a lot of Noise and instead we should take a range of frequency, the idea seems interesting but as stated in the problem in lieu of time we should move forward, I leave this idea to explore in future.

2.0.4 2.4 Can Volatility help?

The Idea was to explore this idea we see Volatility correlation when market is in fear or we say correlation increases as vol increase so this may create unnecessary noise in the data, as we want pure correlation between stocks among sector but when vol increases for the whole market every stock becomes correlated and it may harm data points

```
[ ]: volatility_index = returns_index.rolling(window=30).std().fillna(0)
      volatility_stock = returns_stock.rolling(window=30).std().fillna(0)
```

```
[ ]: for i in range(100):
      corr = np.corrcoef((returns_stock.iloc[:,i]), volatility_stock.iloc[:,i])[0, 1]
      ↪ if corr > 0.1:
          print(corr)
```

No Output, as We have correlation, around 0.002, but as we know correlation takes into account the sign and -ve returns are harming the potential results, we can go to other robust correlation methods. Here I am using Absolute values of returns, which is not so correct but on large dataset it won't harm

```
[ ]: corr_list = []
for i in range(100):
    corr = np.corrcoef(abs(returns_stock.iloc[:,i]), volatility_stock.iloc[:,i])[0, 1]
    if corr > 0.1:
        corr_list.append(corr)
print(len(corr_list))
```

100

Now the idea is **whenever index were very volatile we drop that day to compute correlation matrix and hence we avoid the days when all sectors were correlated**

```
[ ]: cutoff = volatility_index.quantile(0.20, axis=0)
sums = (volatility_index >= cutoff).sum(axis=1)
top_20_perc = volatility_index.index[sums > (volatility_index.shape[1] / 3)]
```

```
[ ]: excluded = returns_stock.drop(top_20_perc)

# Compute the correlation matrix
new_corr_matrix = excluded.corr()
```

Not including further computations here as it will be a mess, **but the No. of clusters didnot changed nor did the classifications**, possible reason is the dataset is huge so in the long run the effect of such events fades away!

3 3. Get the Functional form

```
[ ]: Sector_0 = [5, 13, 15, 24, 31, 34, 35, 37, 48, 56, 61, 62, 64, 65, 68, 70, 76, 83, 85, 86, 89, 93, 94, 95, 97]
Sector_1 = [8, 12, 27, 29, 30, 36, 40, 41, 44, 46, 50, 52, 55, 57, 59, 69, 71, 73, 74, 77, 80, 84, 92, 96, 99]
Sector_2 = [2, 3, 6, 7, 9, 16, 17, 19, 23, 28, 32, 33, 42, 47, 51, 53, 58, 60, 63, 66, 72, 81, 87, 91, 98]
Sector_3 = [0, 1, 4, 10, 11, 14, 18, 20, 21, 22, 25, 26, 38, 39, 43, 45, 49, 54, 67, 75, 78, 79, 82, 88, 90]
sectors = [Sector_0, Sector_1, Sector_2, Sector_3]

sector_names = ['Sector_0', 'Sector_1', 'Sector_2', 'Sector_3']
```

```
[ ]: # OLS Regression

def check_covariance(sector, index, returns_stock, returns_index):
```

```

train_x = returns_stock.iloc[5000:100000, sector ]
train_y = returns_index.iloc[5000:100000, index]

test_x = returns_stock.iloc[100000:200000, sector ]
test_y = returns_index.iloc[100000:200000, index]

ols_final = sm.OLS(train_y, sm.add_constant(train_x)).fit()

# Compute test R2 and test mean squared error
ols_pred = ols_final.predict(sm.add_constant(test_x))
ols_pred = pd.DataFrame(ols_pred, columns=["ols_p"])
ols_actual = test_y

ols_rss = np.sum(np.power(ols_pred.ols_p - ols_actual, 2))
ols_tss = np.sum(np.power(ols_actual - np.mean(ols_actual), 2))
ols_rsqr = 1 - (ols_rss / ols_tss)
# print("\n OLS_R2", ols_rsqr)

ols_MSE = np.sqrt(ols_rss / len(test_y))
# print(" OLS_SME", ols_MSE)

x = ols_final.predict(sm.add_constant(test_x))
y = test_y

corr = np.corrcoef(x, y)[0, 1]

return corr, x

```

```

[ ]: corr_dict = {}

sector_dict = {
    'Sector_0': Sector_0,
    'Sector_1': Sector_1,
    'Sector_2': Sector_2,
    'Sector_3': Sector_3,
    'Sector_4': Sector_4,
    'Sector_5': Sector_5,
    'Sector_6': Sector_6,
    'Sector_7': Sector_7
}

for sector_index in r_index:
    if sector_index:
        sector_name = sector_index[0]
        index = sector_index[1]

```

```

        sector_list = sector_dict[sector_name]
        corr = check_covariance(sector_list, index, returns_stock,
↪returns_index)
        corr_dict[tuple(sector_index)] = corr

for key, value in corr_dict.items():
    print(key, value)

```

```

('Sector_7', 0) 0.40204417297927175
('Sector_4', 1) 0.30425278442714737
('Sector_7', 2) 0.20083661354964677
('Sector_5', 3) 0.2868239099593076
('Sector_2', 4) 0.40991735139618285
('Sector_1', 5) 0.20174049593765148
('Sector_5', 6) 0.43429976064863024
('Sector_0', 7) 0.250788462759403
('Sector_5', 8) 0.3220364392607265
('Sector_2', 9) 0.30951645306415565
('Sector_4', 10) 0.30720682859196574
('Sector_6', 14) 0.20563694917113223

```

What we notice is index 0,4,6 have high adjusted R^2 you notice these were the special indices, and what we also saw was 0,4,6 are the indexes which follow normal distribution suggesting others will fit probably a non-linear curve better

With this simple OLS model we see which sector may be able to explain the indices better, or which indices can have functional form for a particular sector. **We notice our special Indices are not having good R squared as we saw from their distribution was bimodal they may follow a quadratic fit!**

Now that we have best explaining sector for each index lets see if we were able to do a good job

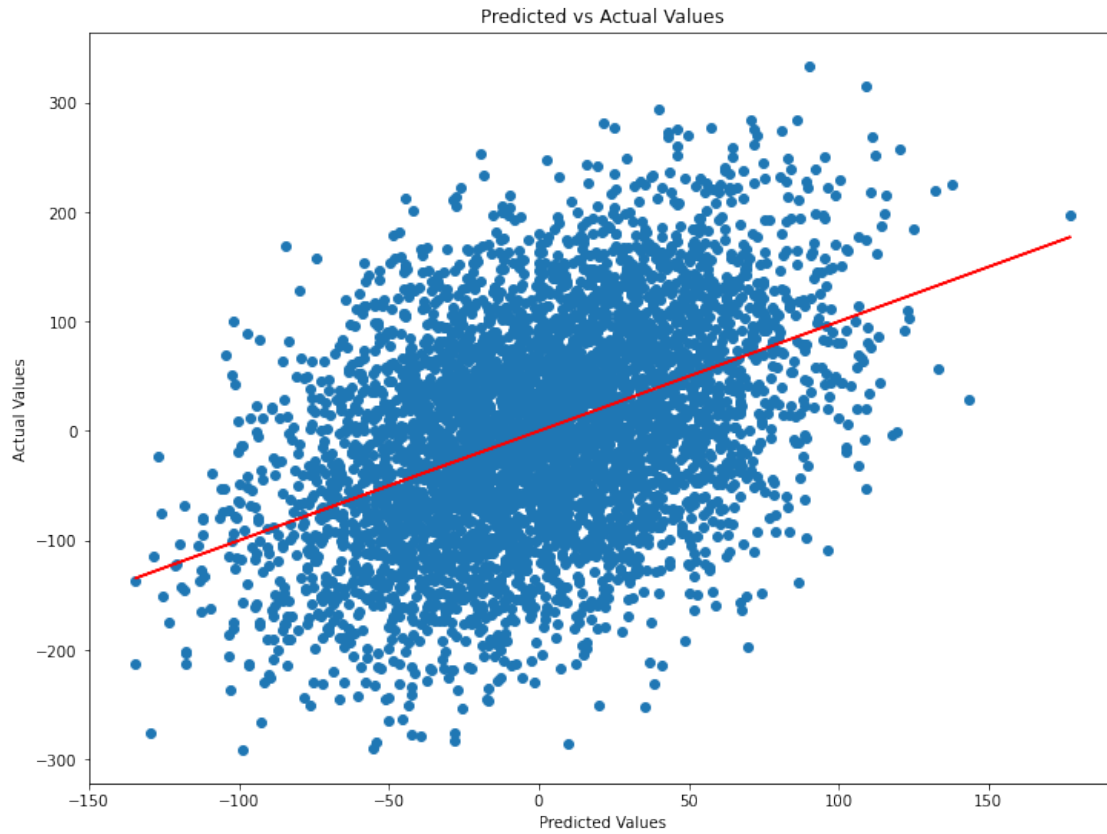
```

[ ]: # scatter plot of predicted vs actual values
plt.scatter(x, y)

# add a line showing the relationship between predicted and actual values
plt.plot(x, x, color='red')

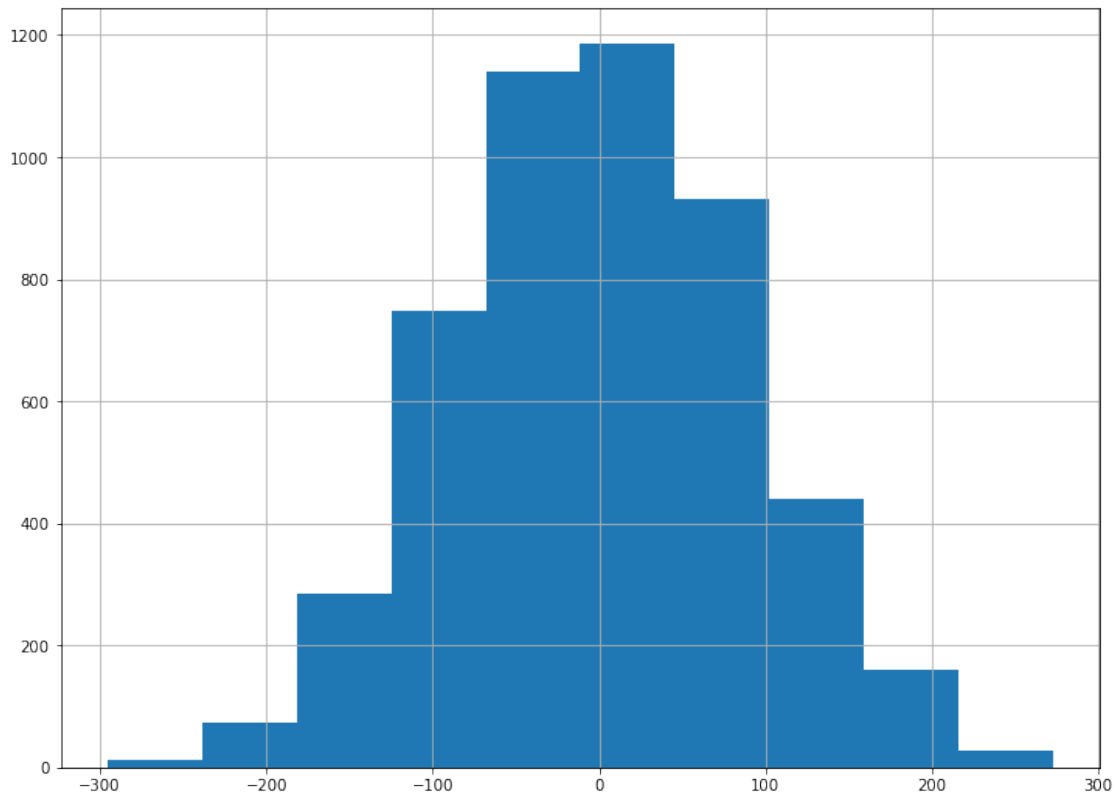
# set axis labels and title
plt.xlabel('Predicted Values')
plt.ylabel('Actual Values')
plt.title('Predicted vs Actual Values')
plt.show()

```

```
[ ]: (y-x).hist();
```

```
[ ]: <AxesSubplot:>
```



Histogram of residuals shows normal distribution indicates that We were able to use all the information from the data! for Index 0, but what about special indexes? but for index 0 we conclude the outlier detection models won't improve the r squared let's see

```
[ ]: #OLS:

train_x = returns_stock.iloc[5000:20000, Sector_5 ]
train_y = returns_index.iloc[5000:20000, 6]

test_x = returns_stock.iloc[:5000, Sector_5 ]
test_y = returns_index.iloc[:5000, 6]

# OLS Regression
ols_final = sm.OLS(train_y, sm.add_constant(train_x)).fit()

# Compute test R^2 and test mean squared error
ols_pred = ols_final.predict(sm.add_constant(test_x))
ols_pred = pd.DataFrame(ols_pred, columns=["ols_p"])
ols_actual = test_y

ols_rss = np.sum(np.power(ols_pred.ols_p - ols_actual, 2))
```

```

ols_tss = np.sum(np.power(ols_actual - np.mean(ols_actual), 2))
ols_rsqr = 1 - (ols_rss / ols_tss)
print("\n OLS_R^2", ols_rsqr)

ols_MSE = np.sqrt(ols_rss / len(test_x))
print(" OLS_SME", ols_MSE)

# Ridge Regression
# generate a sequence of lambdas to try
lambdas = [np.power(10, i) for i in np.arange(4, -4, -0.1)]
alphas = lambdas

# Scale
# train_x_scale = scale(train_x) #In case you want to scale the variables.

# Use 10-fold Cross Validation to find optimal lambda
ridge_cv = RidgeCV(alphas=alphas, cv=10, scoring="neg_mean_squared_error")
ridge_cv.fit(train_x, train_y)

# Build final ridge regression model
ridge_final = Ridge(alpha=ridge_cv.alpha_, fit_intercept=True)
ridge_final.fit(train_x, train_y)

# R squared formula and mean squared error
ridge_pred = ridge_final.predict(test_x)
ridge_actual = test_y
ridge_rss = np.sum(np.power(ridge_pred - ridge_actual, 2))
ridge_tss = np.sum(np.power(ridge_actual - np.mean(ridge_actual), 2))
ridge_rsqr = 1 - ridge_rss / ridge_tss
print("\n Ridge_R^2", ridge_rsqr)

ridge_MSE = np.sqrt(ridge_rss / len(test_x))
print("Ridge_SME", ridge_MSE)

# LASSO Regression

# generate a sequence of lambdas to try
lambdas = [np.power(10, i) for i in np.arange(6, -6, -0.1)]

# Compile model
lasso_cv = LassoCV(cv=10, alphas=lambdas)
lasso_cv.fit(train_x, train_y) # Fit Model

# Build final LASSO regression model
lasso_final = Lasso(alpha=lasso_cv.alpha_, fit_intercept=True)
lasso_final.fit(train_x, train_y)

```

```

# R squared formula and mean squared error
lasso_pred = lasso_final.predict(test_x)
lasso_actual = test_y
lasso_rss = np.sum(np.power(lasso_pred - lasso_actual, 2))
lasso_tss = np.sum(np.power(lasso_actual - np.mean(lasso_actual), 2))
lasso_rsqr = 1 - lasso_rss / lasso_tss
print("\n LASSO_R^2: ", lasso_rsqr)

lasso_MSE = np.sqrt(lasso_rss / len(test_x))
print("LASSO_SME: ", lasso_MSE)

```

```

OLS_R^2 0.18854365722305466
OLS_SME 87.40656291834547

```

```

Ridge_R^2 0.1885435568035969
Ridge_SME 87.40656832671985

```

```

LASSO_R^2: 0.1885428705718849
LASSO_SME: 87.40660528566376

```

```

[ ]: OLS_df = pd.DataFrame(ols_final.summary2().tables[1]["Coef."]).
      ↪rename(columns={"Coef.": "OLS"})
OLS_df.index = ["Intercept",3, 6, 7, 9, 16, 17, 19, 23, 28, 32, 33, 42, 47, 51,
      ↪53, 58, 60, 63]

Ridge_df = pd.DataFrame(
    np.insert(ridge_final.coef_, 0, ridge_final.intercept_),
    index= ["Intercept",3, 6, 7, 9, 16, 17, 19, 23, 28, 32, 33, 42, 47, 51, 53,
    ↪58, 60, 63],
    columns=["Ridge"],)

Lasso_df = pd.DataFrame(
    np.insert(lasso_final.coef_, 0, lasso_final.intercept_),
    index=["Intercept",3, 6, 7, 9, 16, 17, 19, 23, 28, 32, 33, 42, 47, 51, 53,
    ↪58, 60, 63],
    columns=["Lasso"],
)

df = OLS_df.merge(Ridge_df, left_index=True, right_index=True)
df = df.merge(Lasso_df, left_index=True, right_index=True)

df.append(
    pd.DataFrame(
        {
            "OLS": [ols_rsqr, ols_MSE],
            "Ridge": [ridge_rsqr, ridge_MSE],

```

```

        "Lasso": [lasso_rsqu, lasso_MSE],
    },
    index=["R sq", "Mean Sq. Err"],
),
ignore_index=False,
)

```

```

[ ]:
      OLS      Ridge      Lasso
Intercept -0.996119 -0.996115 -0.996063
3          0.072563  0.072562  0.072553
6          0.047825  0.047825  0.047816
7          0.051135  0.051135  0.051133
9          0.071319  0.071315  0.071285
16         0.031397  0.031399  0.031397
17         0.067067  0.067065  0.067048
19         0.076927  0.076920  0.076871
23         0.057811  0.057809  0.057783
28         0.055770  0.055769  0.055759
32         0.041866  0.041867  0.041866
33         0.074638  0.074630  0.074567
42         0.042412  0.042413  0.042412
47         0.059430  0.059429  0.059417
51         0.060064  0.060063  0.060052
53         0.076216  0.076213  0.076193
58         0.074409  0.074401  0.074344
60         0.069084  0.069082  0.069064
63         0.064605  0.064604  0.064586
R sq       0.188544  0.188544  0.188543
Mean Sq. Err 87.406563 87.406568 87.406605

```

So we have all the coefficients and an Intercept for our functional form for Index 7 explained by Sector 5, and similarly we store for all of them and their predictions now!

```

[ ]: predictions.to_csv("predictions_OLS.csv")

```

```

[ ]: df_predictions = pd.read_csv("predictions_OLS.csv", index_col= 'Unnamed: 0')
df_predictions

```

```

[ ]:
      0          1          2          3          4          5  \
100001  14.599583 -2.868377 -11.218019 -10.659914  49.083123 -9.721550
100002   9.429706 -35.651850  12.513619  24.191307 -4.739676  20.456032
100003  63.614170 -31.694544 -49.280298 -34.347178 -12.419088 -10.662190
100004  45.224822 -6.025127 -26.456619 -58.200514  70.550851 -7.282417
100005  43.966904  57.498722 -13.948657 -35.609277 -60.097190 -55.778331
...
199995  52.849782  18.790479   1.358003  71.044418 -9.458841 -25.852707
199996  32.271568 -4.176340 -27.472346  66.621942  23.872060 -35.353183

```

```

199997 -18.895914 -23.932701 -0.135992  9.752798 -38.035320 -3.321031
199998  18.455228 -14.580303 -2.328382 -23.805353 -39.529998 21.130877
199999  14.200704 -15.960729  2.297821 -1.112284  18.581018 -0.301367

```

```

          6          7          8          9          10          11  \
100001  8.634447  5.381625  0.491627 -6.123677 -52.280460  6.136776
100002 -37.912675  2.538937 -24.327310  3.325710  10.458176 -1.031368
100003 -29.694853 40.746073 -12.206510 22.031224  46.142682 -2.707917
100004  60.489385 33.130847 37.745020 19.134000  33.286838  2.263658
100005  19.142217 16.796243 12.348202 79.731543 -17.329954  1.494280
...
199995 -46.854888 28.124784 -28.705011 -21.680873  3.157729 -1.524393
199996 -37.089869 22.809914 -24.286472 49.458189  39.101951 -0.023425
199997 -36.094673 -9.354447 -28.284815 72.338899  8.050806  4.150536
199998 -25.746564 13.658382 -11.886517 26.156238 -46.271405  1.784112
199999  57.916202 12.845423 41.554940 -48.683396 -6.331413 -2.602827

```

```

          14
100001 -50.880467
100002  0.108155
100003  6.093103
100004 -24.189271
100005 -7.897371
...
199995 31.790769
199996 -11.698417
199997  9.338779
199998 15.003824
199999 19.504221

```

[99999 rows x 13 columns]

```

[ ]: corr_index = df_predictions.corr()
      corr_index

```

```

[ ]:
      0          1          2          3          4          5          6  \
0  1.000000  0.168604 -0.646803 -0.100723  0.201042 -0.129139  0.189878
1  0.168604  1.000000 -0.095339 -0.101530  0.180111 -0.639282  0.195175
2 -0.646803 -0.095339  1.000000  0.060571 -0.117424  0.071206 -0.091976
3 -0.100723 -0.101530  0.060571  1.000000 -0.082980  0.077914 -0.479793
4  0.201042  0.180111 -0.117424 -0.082980  1.000000 -0.135318  0.173270
5 -0.129139 -0.639282  0.071206  0.077914 -0.135318  1.000000 -0.142724
6  0.189878  0.195175 -0.091976 -0.479793  0.173270 -0.142724  1.000000
7  0.981538  0.169192 -0.605527 -0.091527  0.199075 -0.131094  0.190376
8  0.185517  0.192679 -0.089878 -0.481038  0.174610 -0.140630  0.983335
9 -0.114321 -0.121983  0.071032  0.060119 -0.484733  0.092329 -0.110824
10 -0.058610 -0.468122  0.033385  0.042092 -0.061847  0.119316 -0.086869

```

```

11  0.057869  0.047617 -0.033446 -0.023067  0.275722 -0.036387  0.050152
14 -0.125274 -0.108240  0.069682  0.036310 -0.664185  0.080764 -0.114421

```

```

          7          8          9          10          11          14
0   0.981538  0.185517 -0.114321 -0.058610  0.057869 -0.125274
1   0.169192  0.192679 -0.121983 -0.468122  0.047617 -0.108240
2  -0.605527 -0.089878  0.071032  0.033385 -0.033446  0.069682
3  -0.091527 -0.481038  0.060119  0.042092 -0.023067  0.036310
4   0.199075  0.174610 -0.484733 -0.061847  0.275722 -0.664185
5  -0.131094 -0.140630  0.092329  0.119316 -0.036387  0.080764
6   0.190376  0.983335 -0.110824 -0.086869  0.050152 -0.114421
7   1.000000  0.185892 -0.114207 -0.059675  0.056856 -0.123086
8   0.185892  1.000000 -0.109910 -0.087189  0.049711 -0.115790
9  -0.114207 -0.109910  1.000000  0.051124 -0.061687  0.131941
10 -0.059675 -0.087189  0.051124  1.000000 -0.012146  0.032760
11  0.056856  0.049711 -0.061687 -0.012146  1.000000 -0.444017
14 -0.123086 -0.115790  0.131941  0.032760 -0.444017  1.000000

```

3.1 Let's try NN to see weather we have better fits in higher dimension

Idea is if we are able to achieve better fits for some indexes we will try to fit non-linear forms to that particular index wherever the **OLS** performed bad and **NN** was better

Also on observation I found if we give volatility of stocks as a feature along with returns we can achieve a better fits, but we will loose explainibility, But still the improvement was significant so I went with Volatility as a feature too!

```

[ ]: import tensorflow as tf
import numpy as np
from tensorflow.keras import regularizers

index_sector = [3,0,3,2,1,0,2,3,2,1,0,3,1,2,1]
sectors = [Sector_0,Sector_1,Sector_2,Sector_3]

for i in range(1,15):

    s = index_sector[i]
    sector = sectors[s]

    train_x = pd.concat([returns_stock.iloc[50:50000, sector],
↪volatility_stock.iloc[50:50000, sector]], axis=1)
    train_y = returns_index.iloc[50:50000, i]

    # Define the neural network architecture
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, activation='relu', input_shape=(train_x.
↪shape[1],),kernel_regularizer=regularizers.l2(0.01)),

```

```

        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(32,
↪activation='relu', kernel_regularizer=regularizers.l2(0.01)),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(1)
    ])

    # Compile the model
    model.compile(optimizer='adam', loss='mean_absolute_error')

    # Train the model
    model.fit(train_x, train_y, epochs=100, batch_size=32, verbose=0)

    # Make predictions on new data
    test_x = pd.concat([returns_stock.iloc[50000:190000, sector],
↪volatility_stock.iloc[50000:190000, sector]], axis=1)
    test_y = returns_index.iloc[50000:190000, i]
    prediction = model.predict(test_x)

    # Append predictions to list
    predictions_df2.append(prediction.flatten())

    corr = np.corrcoef(prediction.flatten(), test_y)[0, 1]

    print("index", i)
    print("Sector", s)
    print("corr", corr)

    # Add corr value to corr_dict with sector s as key
    if s not in corr_dict2:
        corr_dict2[s] = []
    corr_dict2[s].append(corr)

# Create dataframe from predictions list
predictions_df = pd.DataFrame(predictions_df2).transpose()

```

So we indeed see better fits for indexes that were not able to fit via OLS specially for our special indexes the correlation reached to as good as 60%

```

[ ]: predictions_df.to_csv("predictions_using_vol.csv_0134567891011121314",
↪index=False)

```

While training I lost the index 2 so i decided to leave him behind as I didn't have time to do it again

```

[ ]: predictions_df

```



```

[ ]:
      0      1      2      3      4      5  \
0      17.594763 -5.907107 -41.906338 -31.463125  63.326897  46.584667
1      12.247170 -52.665287  4.143923 -42.779728  126.282486 -36.071136
2      46.177990  13.995529  85.398790  45.671730  10.235013  12.134693
3     -16.572327 -9.301745  4.143923  8.495736  121.733116 -44.318356
4     -10.954910 -34.153374  6.393514 -10.273191  125.194040 -93.087320
...
139995 -4.900393  25.528957  9.251000 -6.012039  38.088190 -53.646770
139996  46.831745 -5.095889  6.647972  32.627070  39.171944 -17.196499
139997  38.983707  91.675160  63.899853  45.233627  33.059624  18.747885
139998 -42.178562  18.908297  5.268075 -81.588326  5.752670 -21.569012
139999  50.115322 -9.587955  95.820380  7.427365 -12.094053  32.542150

      6      7      8      9      10      11  \
0      39.679550  0.086075  10.045840  7.147789 -9.416501  10.561757
1     -9.598078 -4.595795  7.403167  6.773143 -33.981880  15.748678
2      14.021536  9.902845  29.352910  34.357900  27.667934  53.057940
3     -63.041490 -1.520072 -11.133469  7.340301 -30.576363 -52.403263
4      23.836360 -68.631030  6.233618  7.072279 -35.366510 -45.121735
...
139995  22.751050 -31.991814 -2.113419  20.685860 -34.199657 -38.219707
139996 -20.002010 -6.774857  30.088495 -138.241800  16.342909  11.404474
139997 -16.684101 -5.815474 -41.439890 -127.795210  23.659883  18.779894
139998 -27.561820 -41.875580  6.749308  6.008812  23.403837  56.787205
139999  57.881330 -14.308416  16.358028  10.395222  34.054146 -1.540191

      12      13
0      17.711887 -118.660600
1       7.835031  119.730040
2     -36.433144  38.070065
3      17.191221  51.472550
4      59.949900  43.690270
...
139995  29.161823 -71.722626
139996  2.914329  48.649580
139997 -13.325199  31.962614
139998 -30.369127 151.525910
139999  13.080715  32.240562

```

[140000 rows x 14 columns]

3.2 5. Trading Strategy Using Portfolio Optimization

Now that we have Cov of Indexes and Predictions we will use a technique that optimizes a portfolio using the **Mean-Variance optimization model with a zero-sum constraint**. It takes as input the expected returns, covariance matrix, target volatility, and risk-free rate of the assets in the portfolio, and returns the optimal weights for each asset.

```
[ ]: def optimize_portfolio(expected_returns, cov_matrix, target_volatility,
    ↪risk_free_rate):

    n_assets = len(cov_matrix)

    # Define optimization variables
    w = cp.Variable(n_assets)

    # Define objective function
    objective = cp.Maximize(cp.sum(w.T @ expected_returns) - risk_free_rate)

    # Define constraints
    constraints = [cp.sum(w) == 0,
                   cp.quad_form(w, cov_matrix) <= target_volatility**2]

    # Solve the optimization problem
    problem = cp.Problem(objective, constraints)
    problem.solve()

    # Retrieve the optimal weights
    optimal_weights = w.value

    return optimal_weights

# Test the function
expected_returns = df_predictions.iloc[:50000].values
cov_matrix = corr_index.values

# Set the target volatility and risk-free rate
target_volatility = 0.1
risk_free_rate = 0.03

# Optimize the portfolio
optimal_weights = optimize_portfolio(expected_returns, cov_matrix,
    ↪target_volatility, risk_free_rate)

print(optimal_weights)
```

Assumptions:

The returns on assets are normally distributed. (For Predictions from NN Yes they are!) The returns on assets are linearly related to each other. The expected returns and covariance matrix used in the optimization are accurate and reliable.

```
[ ]: expected_returns = df_predictions.iloc[:50000,]
    cov_matrix = corr_index

    # Set the target volatility and risk-free rate
```

```
target_volatility = 0.1
risk_free_rate = 0.03

# Optimize the portfolio
optimal_weights = optimize_portfolio(expected_returns, cov_matrix,
    ↪target_volatility, risk_free_rate)

print(optimal_weights)
```

3.2.1 Did not get enough time to train and Submit the Results but Will do it in Future!! Had a great time exploring the Quant-Research type of job!