report

July 12, 2024

1 Global Stock Market Analytics

in this project we will look at predicting the open direction of the Nifty 50 index by looking at other indices and indicators. We will break the project up into five phases:

1. preparing the master data from the global indices
2. preliminary analysis of data
3. predictive modelling of open direction of Nifty 50
4. comparing different models at prediction
5. sentiment analysis of X / Twitter data relating to Nifty 50

1.1 Phase 1 - Prepare the Master Data

We begin by preparing the data that we will be working with. We import the libraries we will be using, set some plotting configurations, and declare some constants. The indexes of interest are:

- NSEI: Nifty 50
- DJI: Dow Jones Index
- IXIC: Nasdaq
- HSI: Hang Seng
- N225: Nikkei 225
- GDAXI: Dax
- VIX: Volatility Index

[ ]:
```
import os

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import statsmodels.api as sm
import yfinance as yf
import ta

import nltk

from scipy import stats
```

```python
from statsmodels.api import Logit
from statsmodels.stats.outliers_influence import variance_inflation_factor

from sklearn.metrics import classification_report, roc_curve, roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler, StandardScaler

import torch
import torch.nn as nn
import torch.utils.data as utils

from string import punctuation, digits

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.sentiment.vader import SentimentIntensityAnalyzer

from wordcloud import WordCloud



plt.style.use("ggplot")
sns.set_style("darkgrid")
sns.set_context("paper")


INDICES  = ['NSEI', 'DJI', 'IXIC', 'HSI', 'N225', 'GDAXI', 'VIX']
COLUMNS  = [f"{index}_DAILY_RETURNS" for index in INDICES]
```

Next, we declare a function that we will use to download the OHLC data, and make use of it:

[ ]:
```python
def retrieve_data(index, start_date = '2017-12-1', end_date = '2024-1-31',␣
 ↪progress = False):
    data = yf.download(f'^{index}', start_date, end_date, progress = progress)

    # create daily returns for each index
    data['Daily Returns'] = data.Close.pct_change() * 100

    # rename columns - prefix with index name
```

```
    data.columns = ["_".join(c.upper() for c in column.split()) for column in␣
↪data.columns]
    data.columns = [f"{index}_{column}" for column in data.columns]

    return data


data = [retrieve_data(index) for index in INDICES]
```

Next, we declare a function we will use to test the normality of the data we will be working with:

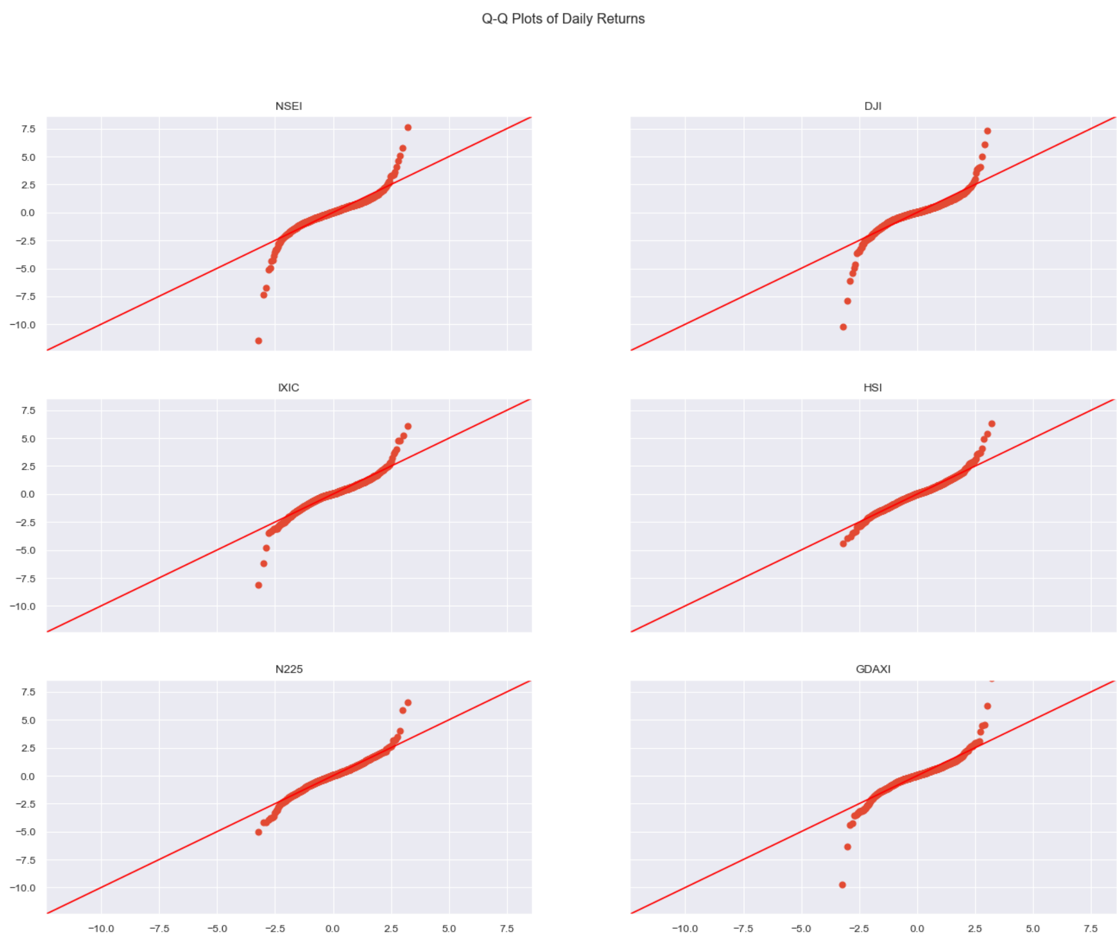
```
[ ]: def test_normality(data, column_name, index_name):
    print()
    print(f"\t Index {index_name}")
    print(f"\tColumn {column_name}")
    print()

    data = data[column_name].dropna()

    if data.shape[0] < 50:
        print("\t     Shapiro-Wilks Test:")
        result = stats.shapiro(data)
    else:
        print("\tKolmogorov-Smirnov Test:")
        result = sm.stats.diagnostic.lilliefors(data)

    print(f"\t                p-value: {result[1]}")

    if result[1] < 0.05:
        print("\treject null hypothesis - data is not drawn from a normal␣
↪distribution")
    else:
        print("\tfail to reject null hypothesis - data is drawn from a normal␣
↪distribution")

    print()


for d, c, i in zip(data, COLUMNS, INDICES):
    test_normality(d, c, i)
```

```
 Index NSEI
Column NSEI_DAILY_RETURNS

Kolmogorov-Smirnov Test:
                p-value: 0.0009999999999998899
reject null hypothesis - data is not drawn from a normal distribution
```

```
 Index DJI
Column DJI_DAILY_RETURNS

Kolmogorov-Smirnov Test:
                p-value: 0.0009999999999998899
reject null hypothesis - data is not drawn from a normal distribution


 Index IXIC
Column IXIC_DAILY_RETURNS

Kolmogorov-Smirnov Test:
                p-value: 0.0009999999999998899
reject null hypothesis - data is not drawn from a normal distribution


 Index HSI
Column HSI_DAILY_RETURNS

Kolmogorov-Smirnov Test:
                p-value: 0.0009999999999998899
reject null hypothesis - data is not drawn from a normal distribution


 Index N225
Column N225_DAILY_RETURNS

Kolmogorov-Smirnov Test:
                p-value: 0.0009999999999998899
reject null hypothesis - data is not drawn from a normal distribution


 Index GDAXI
Column GDAXI_DAILY_RETURNS

Kolmogorov-Smirnov Test:
                p-value: 0.0009999999999998899
reject null hypothesis - data is not drawn from a normal distribution


 Index VIX
Column VIX_DAILY_RETURNS

Kolmogorov-Smirnov Test:
                p-value: 0.0009999999999998899
reject null hypothesis - data is not drawn from a normal distribution
```

Next, we declare a function we will use to create qq-plots of the data we will be working with:

[ ]:
```
#| label: q-q-plots-daily-returns

def qq_plots(data, title, count = 6):
    fig, axes = plt.subplots(3, 2, figsize = (16, 12), sharex = True, sharey =␣
 ↪True)
    fig.suptitle(title)

    for index in range(count):
        axes[index // 2, index % 2].set_title(INDICES[index])
        sm.graphics.qqplot(data[index][COLUMNS[index]].dropna(), line = "45",␣
 ↪fit = True, ax = axes[index // 2, index % 2])
        axes[index // 2, index % 2].set_xlabel("")
        axes[index // 2, index % 2].set_ylabel("")


qq_plots(data, "Q-Q Plots of Daily Returns")
```



The daily returns do not appear to follow - or be drawn from - a Normal Distribution - specifically at the tails.

Next, we declare a function we will use to merge the data, using LOCF for missing data, and adding variables for MONTH, QUARTER, and YEAR:

[ ]:
```
def merge_data(data, start_date = '2018-01-02', end_date = '2023-12-29'):
    # merge data with outer join
    merged = pd.concat(data, axis = 1)

    # impute missing data using LOCF (forward fill)
    merged.ffill(inplace = True)

    # add indicators for MONTH, QUARTER, and YEAR
    merged['MONTH']   = merged.index.month
    merged['QUARTER'] = merged.index.quarter
    merged['YEAR']    = merged.index.year

    return merged[start_date:end_date]


master = merge_data(data)
```

1.2 Phase 2 - Preliminary Analysis

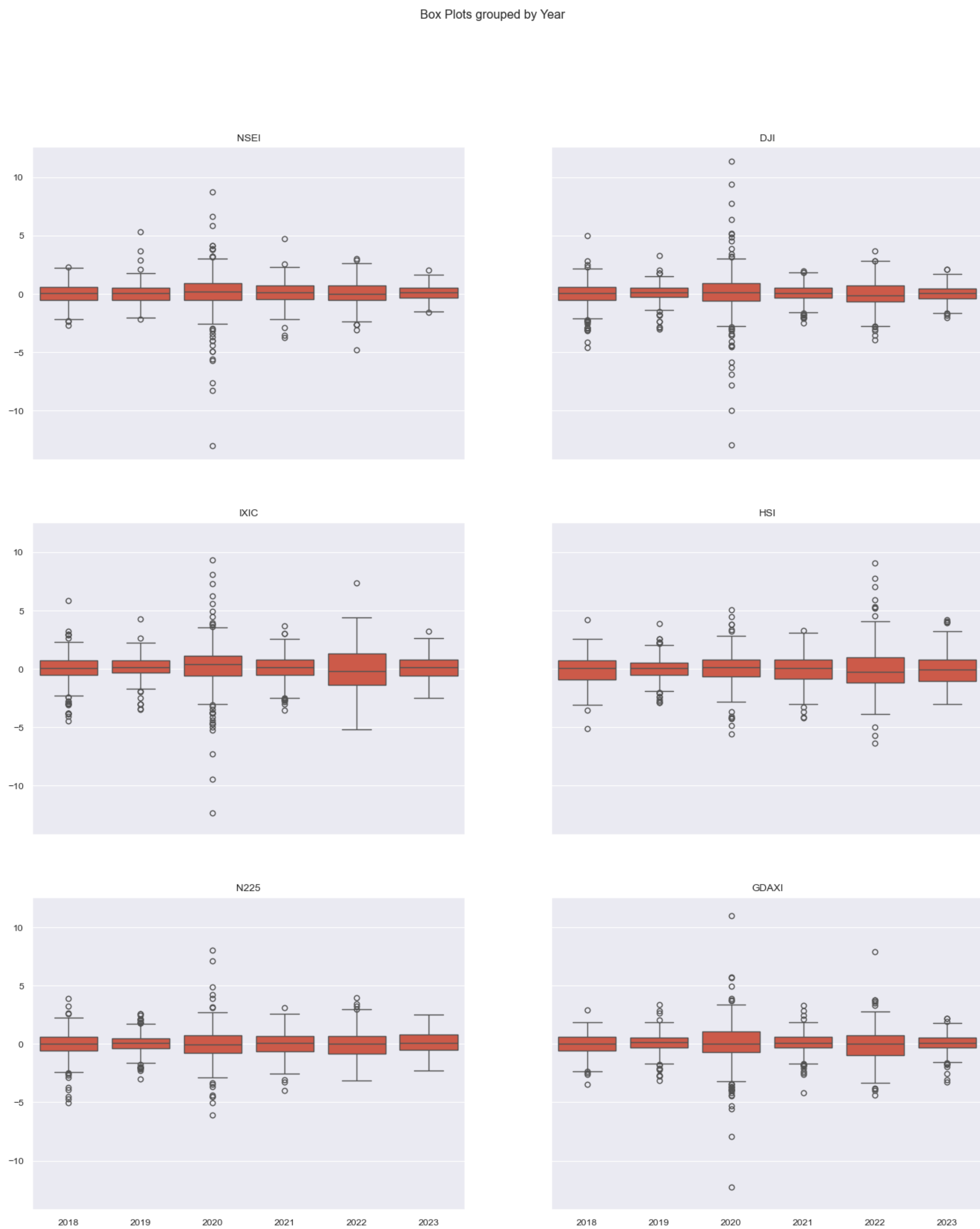
Now that we have our master data we perform some preliminary analysis hoping to answer the following questions:

1. Which index has given consistently good returns?
2. Which index was highly volatile?
3. How are global markets correlated during 6 years period and is the correlation structure similar in the recent year-2023?
4. Assuming primary target variable as "Nifty Opening Price Direction", what are preliminary insights?

Lets define a few functions we will use across our analysis - to begin with, a function that plots box plots of the daily returns by year:

[ ]:
```
#| label: box-plots-daily-returns-yearly

def performance_analytics_box_plots(data, group_by, title, count = 6):
    fig, axes = plt.subplots(3, 2, figsize = (16, 18), sharex = True, sharey =␣
 ↪True)
    fig.suptitle(title)

    for index in range(count):
        axes[index // 2, index % 2].set_title(INDICES[index])
```

```
        sns.boxplot(x = data[group_by], y = data[COLUMNS[index]], ax = axes[index // 2, index % 2])
        axes[index // 2, index % 2].set_xlabel("")
        axes[index // 2, index % 2].set_ylabel("")


performance_analytics_box_plots(master, "YEAR", "Box Plots grouped by Year")
```



All indexes seem pretty consistent - all years have similar spreads, and consistent medians, with one or two exceptions. All indexes for 2020 have more outliers than normal. But HSI seems to have more outliers in 2022 than in 2020.

Next, we declare a function that prints a table of summary statistics for the daily returns of each index:

[ ]:
```
def performance_analytics_tables(data, group_by, count = 6):
    for index in range(count):
        table = round(data.groupby(group_by, observed = False)[COLUMNS[index]].
 ↪agg(['count', 'mean', 'std', 'var']), 3)
        print(f"\n{INDICES[index]}\n\n{table}\n\n")


performance_analytics_tables(master, "YEAR")
```

```
NSEI

      count   mean    std    var
YEAR
2018    260  0.012  0.804  0.647
2019    260  0.062  0.862  0.744
2020    262  0.059  2.004  4.015
2021    261  0.094  0.980  0.960
2022    260  0.055  1.096  1.202
2023    260  0.079  0.620  0.384



DJI

      count   mean    std    var
YEAR
2018    260 -0.035  1.143  1.306
2019    260  0.099  0.784  0.614
2020    262  0.057  2.277  5.186
2021    261  0.075  0.773  0.597
2022    260 -0.025  1.237  1.531
2023    260  0.060  0.709  0.503



IXIC

      count   mean    std    var
```

YEAR				
2018	260	-0.020	1.330	1.768
2019	260	0.133	0.975	0.950
2020	262	0.170	2.200	4.838
2021	261	0.096	1.124	1.262
2022	260	-0.124	2.000	4.001
2023	260	0.157	1.085	1.177

HSI

	count	mean	std	var
YEAR				
2018	260	-0.035	1.244	1.547
2019	260	0.033	0.981	0.962
2020	262	0.026	1.445	2.087
2021	261	-0.028	1.262	1.593
2022	260	-0.021	2.054	4.221
2023	260	-0.053	1.409	1.984

N225

	count	mean	std	var
YEAR				
2018	260	-0.047	1.198	1.436
2019	260	0.065	0.860	0.740
2020	262	0.013	1.615	2.608
2021	261	0.029	1.152	1.328
2022	260	-0.036	1.262	1.592
2023	260	0.105	0.999	0.997

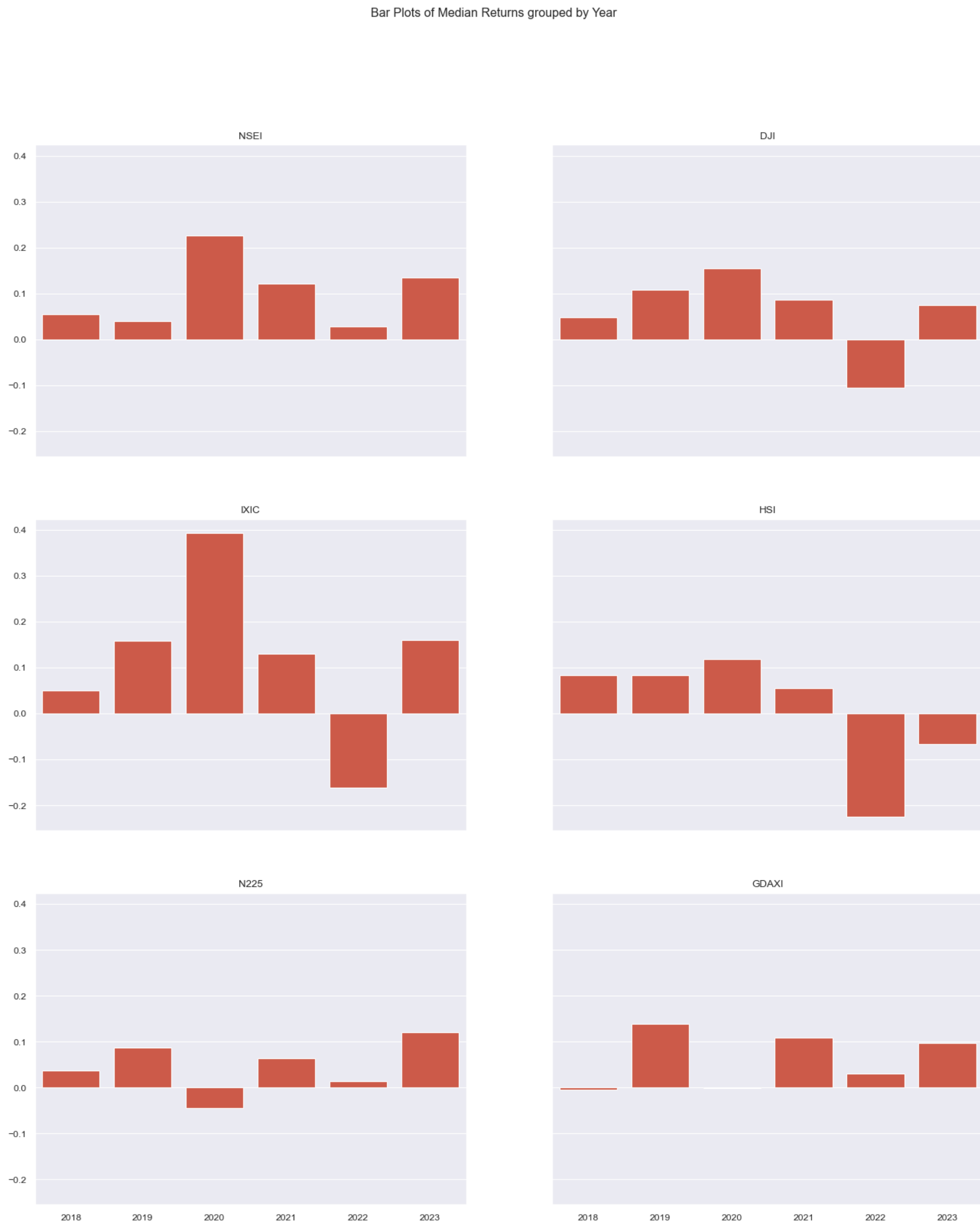
GDAXI

	count	mean	std	var
YEAR				
2018	260	-0.056	0.975	0.951
2019	260	0.084	0.888	0.788
2020	262	0.043	2.064	4.260
2021	261	0.071	0.899	0.808
2022	260	-0.035	1.460	2.133
2023	260	0.082	0.809	0.655

The clear winner here is NSEI - not one year in the range has a negative mean return. And with the exception of 2020 and 2022, NSEI has low volatility (< 1) throughout all years.

Next, we declare a function that plots median daily returns by year:

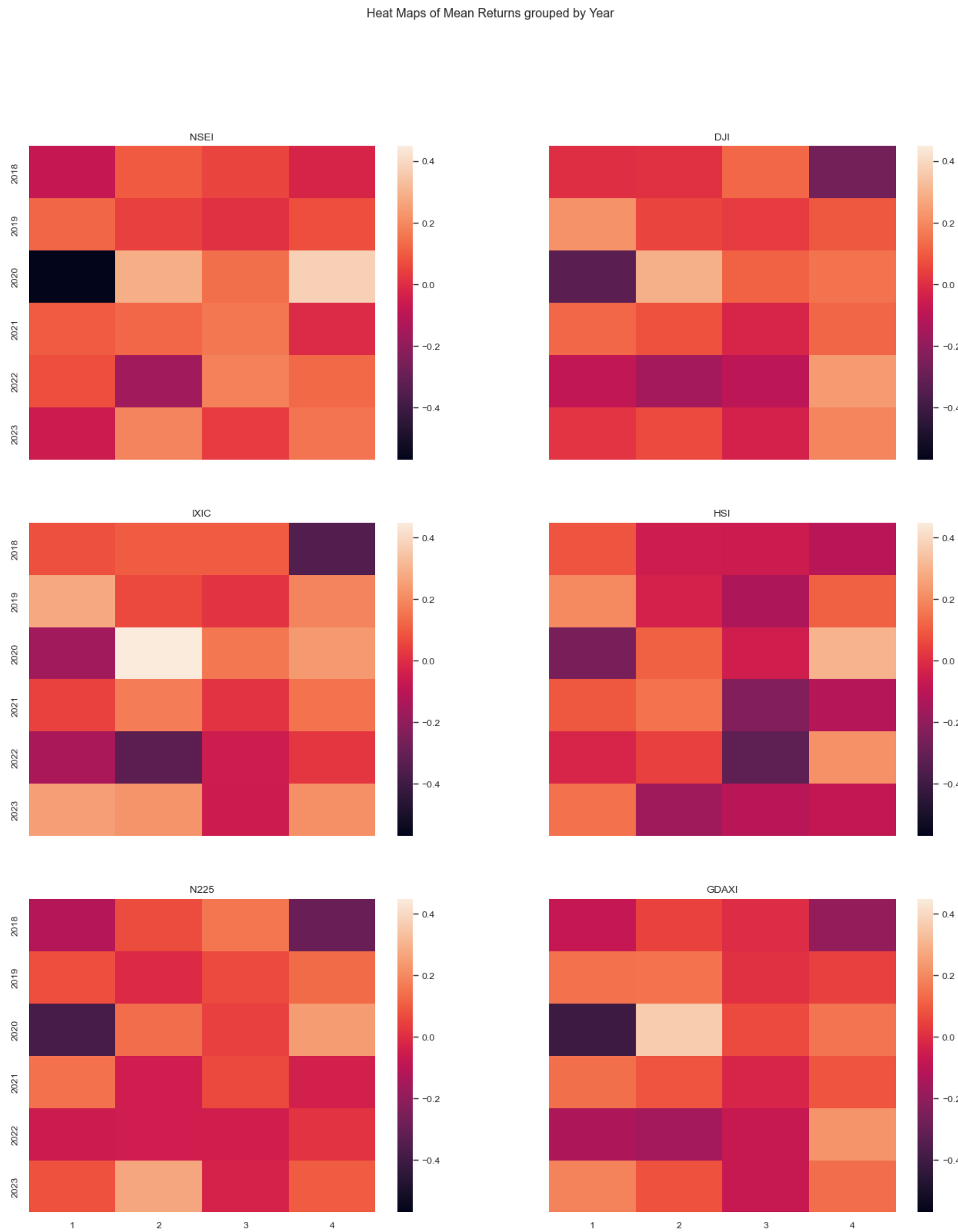
[ ]:
```
#| label: bar-plots-daily-returns-yearly

def performance_analytics_bar_plots(data, group_by, title, count = 6, aggfunc =
 ↪"median"):
    fig, axes = plt.subplots(3, 2, figsize = (16, 18), sharex = True, sharey =
 ↪True)
    fig.suptitle(title)

    for index in range(count):
        axes[index // 2, index % 2].set_title(INDICES[index])
        table = data.groupby(group_by, observed = False)[COLUMNS[index]].
 ↪agg([aggfunc])
        sns.barplot(x = table.index, y = table[aggfunc], ax = axes[index // 2,
 ↪index % 2])
        axes[index // 2, index % 2].set_xlabel("")
        axes[index // 2, index % 2].set_ylabel("")

performance_analytics_bar_plots(master, "YEAR", "Bar Plots of Median Returns
 ↪grouped by Year")
```



The clear winner here is NSEI - at no time is the median daily returns for any of the years below 0. IXIC has an unusually high 2020, but a bad 2022. HSI also has an unusually bad 2022.

Next, we plot of heatmaps by year and quarter showing mean returns:

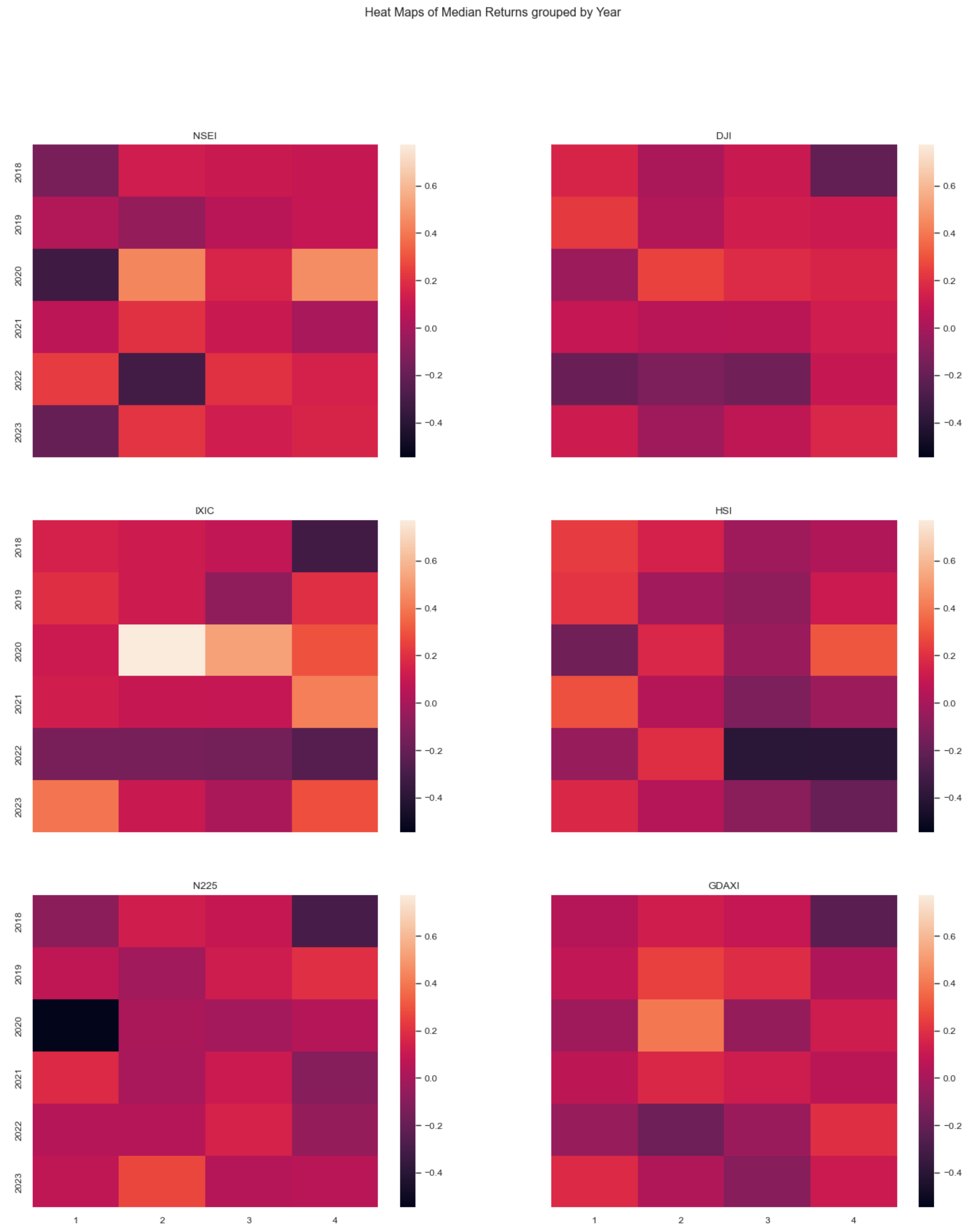
[ ]:
```python
#| label: heat-maps-mean-daily-returns-yearly

def performance_analytics_heat_maps(data, group_by, title, column = "QUARTER",␣
↪count = 6, aggfunc = "median"):
    fig, axes = plt.subplots(3, 2, figsize = (16, 18), sharex = True, sharey =␣
↪True)
    fig.suptitle(title)

    tables = []
    values = []
    for index in range(count):
        table = pd.pivot_table(data, values = COLUMNS[index], index =␣
↪[group_by], columns = [column], aggfunc = aggfunc, observed = False)
        tables.append(table)
        values.extend(table.values.ravel())

    vmax = max(values)
    vmin = min(values)

    for index in range(count):
        axes[index // 2, index % 2].set_title(INDICES[index])
        sns.heatmap(tables[index], ax = axes[index // 2, index % 2], vmin =␣
↪vmin, vmax = vmax)
        axes[index // 2, index % 2].set_xlabel("")
        axes[index // 2, index % 2].set_ylabel("")

performance_analytics_heat_maps(master, "YEAR", "Heat Maps of Mean Returns␣
↪grouped by Year", aggfunc = "mean")
```



With the exception of the 1st quarter in 2020, NSEI has pretty consistent daily returns - where most cells sre pretty bright, denoting above 0. Most of the other indexes have a blend of light and dark, which would indicate more volatile behaviour over the quarters.

Next we look at heatmaps by year and quarter showing median returns:

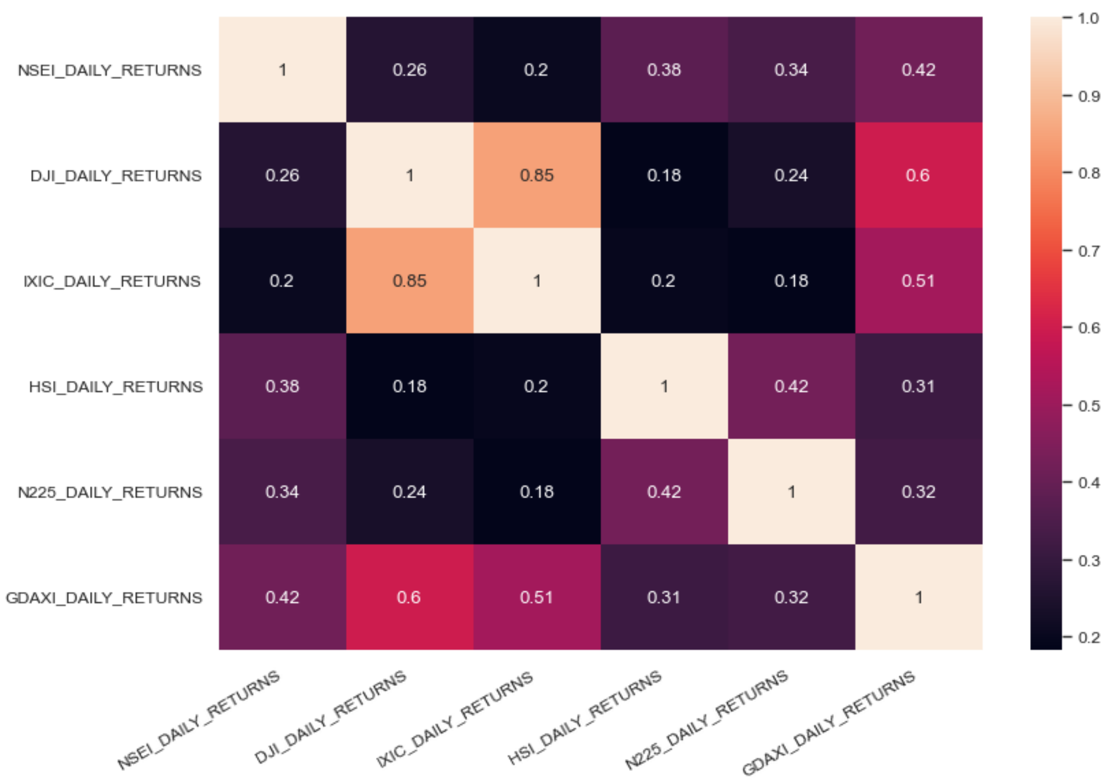
[ ]:
```
#| label: heat-maps-median-daily-returns-yearly

performance_analytics_heat_maps(master, "YEAR", "Heat Maps of Median Returns␣
↪grouped by Year")
```



On the other hand, when looking at median returns across quarters NSEI seems pretty average - there does not seem to be a clear winner here.

Lets look at a correlation matrix of the 6 years daily returns:

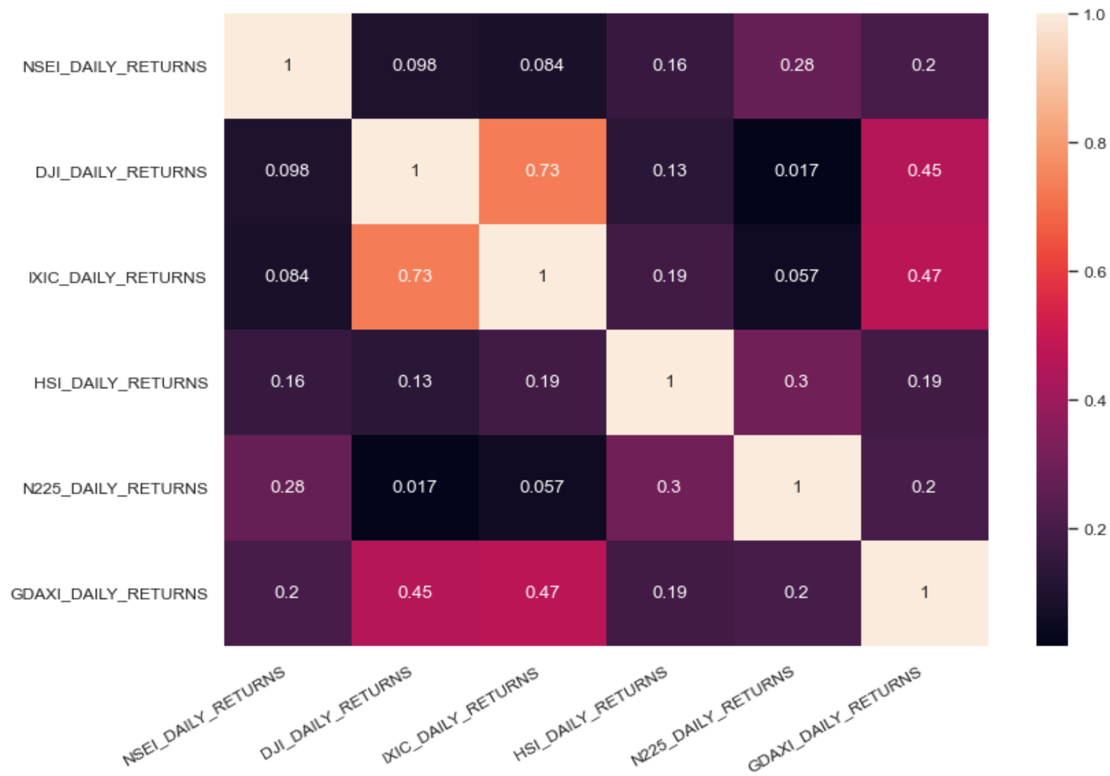
[ ]:
```python
#| label: correlation-matrix-daily-returns-all-years

def correlation_matrix(data):
    plt.figure(figsize = (9, 6))
    matrix = data[COLUMNS[:-1]].corr()

    ax = sns.heatmap(matrix, annot = True)
    ax.set_xticklabels(
        ax.get_xticklabels(),
        rotation = 30,
        horizontalalignment = "right"
    )


correlation_matrix(master)
```



It looks like strong correlation between daily returns of IXIC and DJI, and some correlation between GDAXI and DJI. These indexes are likely to result in multicolinearity at the regression stage.

Next, we look at the same correlation matrix for the 2023 year alone to see if there are similar correlations:

```
[ ]: #| label: correlation-matrix-daily-returns-last-year

correlation_matrix(master['2023-01-02':'2023-12-29'])
```

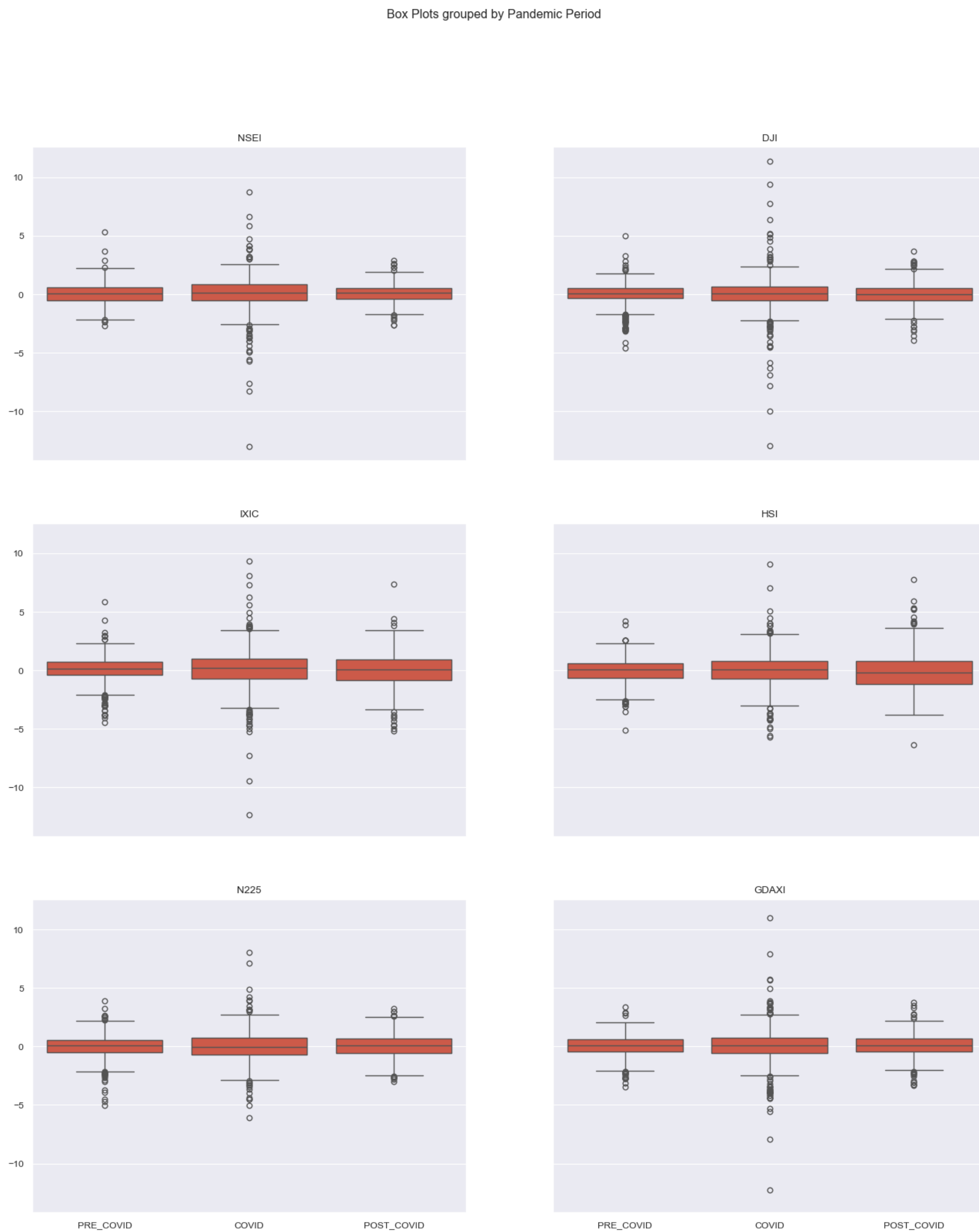


We can see similar - but slightly weaker - correlations exist between the same indexes for 2023.

Next, we do similar analysis for the Pandemic period - Pre Covid, Covid, and Post Covid:

```
[ ]: #| label: box-plots-daily-returns-covid

CONDITIONS = [(master.index <= '2020-01-30'), ('2022-05-05' <= master.index)]
CHOICES    = ['PRE_COVID', 'POST_COVID']

master['PANDEMIC'] = np.select(CONDITIONS, CHOICES, 'COVID')
master['PANDEMIC'] = pd.Categorical(master['PANDEMIC'], categories =
  ↪['PRE_COVID', 'COVID', 'POST_COVID'], ordered = True)
```

```
performance_analytics_box_plots(master, "PANDEMIC", "Box Plots grouped by␣
↪Pandemic Period")
```



We can see that the spreads of each index over the Pandemic are consistent, with the Covid period itself having more outliers - which of course you might expect.

Next we look at summary statistics for the Pandemic:

```
[ ]: performance_analytics_tables(master, "PANDEMIC")
```

NSEI

	count	mean	std	var
PANDEMIC				
PRE_COVID	542	0.033	0.830	0.690
COVID	589	0.069	1.563	2.442
POST_COVID	432	0.082	0.777	0.604

DJI

	count	mean	std	var
PANDEMIC				
PRE_COVID	542	0.034	0.969	0.938
COVID	589	0.043	1.657	2.745
POST_COVID	432	0.038	0.981	0.962

IXIC

	count	mean	std	var
PANDEMIC				
PRE_COVID	542	0.062	1.153	1.330
COVID	589	0.077	1.812	3.285
POST_COVID	432	0.065	1.525	2.326

HSI

	count	mean	std	var
PANDEMIC				
PRE_COVID	542	-0.013	1.126	1.268
COVID	589	0.012	1.508	2.274
POST_COVID	432	-0.047	1.664	2.769

N225

	count	mean	std	var

```
PANDEMIC
PRE_COVID      542  0.000  1.046  1.095
COVID          589  0.017  1.417  2.009
POST_COVID     432  0.055  1.066  1.136
```

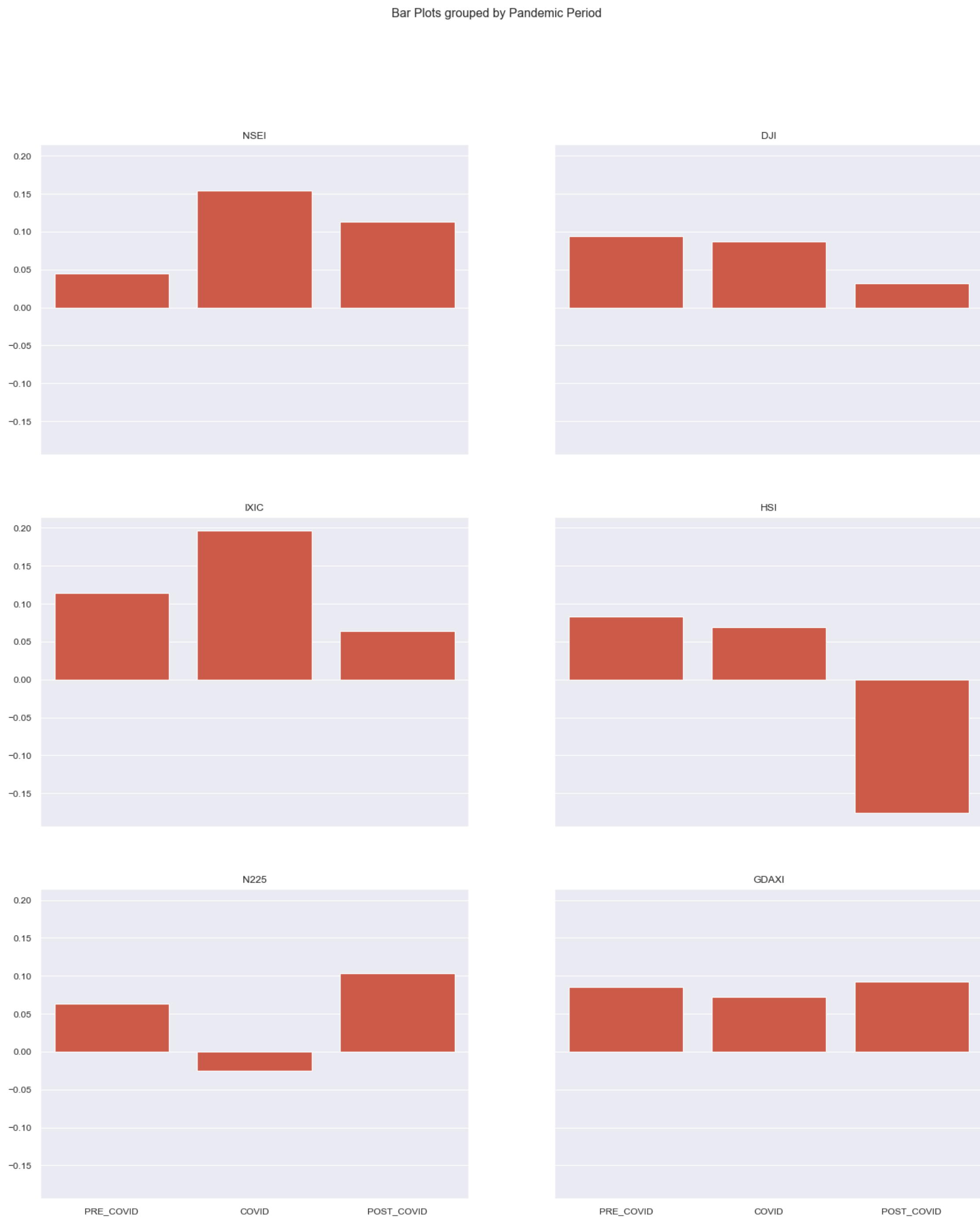
```
GDAXI

            count   mean    std    var
PANDEMIC
PRE_COVID     542  0.011  0.935  0.875
COVID         589  0.035  1.634  2.671
POST_COVID    432  0.052  1.027  1.055
```

All indexes had higher volatility over the Covid period.

- NSEI performed reasonably well over the Covid period, with an increase in volatility in the Covid period, and with a significant bump in the Post Covid period.
- DJI seemed consisten over the three periods, with an increase in volatility in the Covid period.
- IXIC looked pretty good over the three period, but maybe slightly more volatile overall, and in particular in the Covid period.
- HSI has performed poorely in general, with negative returns in the pre and post Covid periods, and with consistently greater volatility than most.
- N225 appears to perform not so well, and with relatively high volatility.
- GDAXI also appears to perform not so well in general, and with relatively high volatility.

Next, we we look at median daily returns by Pandemic period:

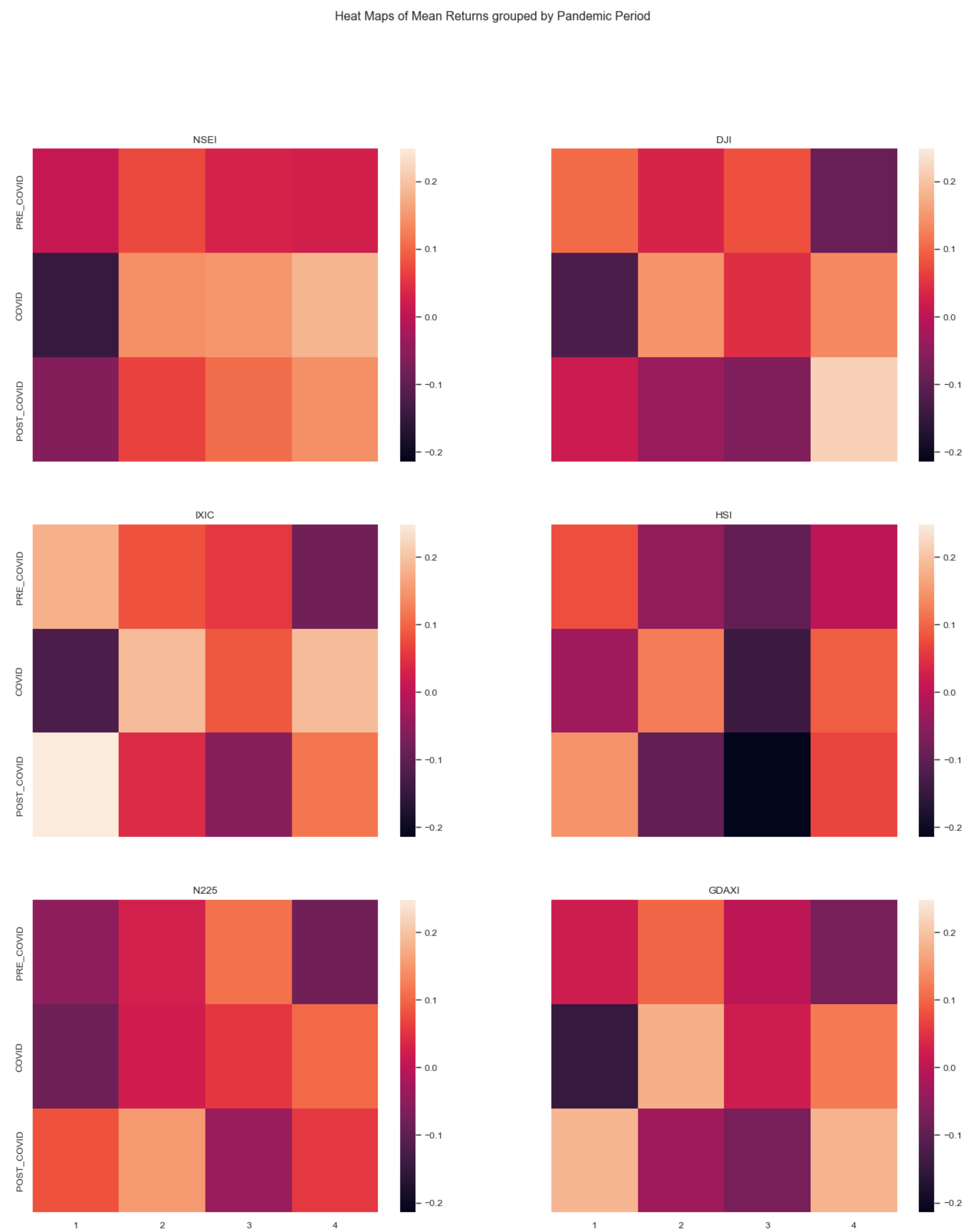
[ ]:
```
#| label: bar-plots-daily-returns-covid

performance_analytics_bar_plots(master, "PANDEMIC", "Bar Plots grouped by␣
↪Pandemic Period")
```



We can see that IXIC looks like the clear winner here, with NSEI in second place, and DJI and GSAXI in a fight for third place. HSI appears to have had a terrible Post Covid period, and N225 appears to have had a pretty bad Covid period.

Next, we plot of heatmaps by Pandemic period and quarter showing mean returns:

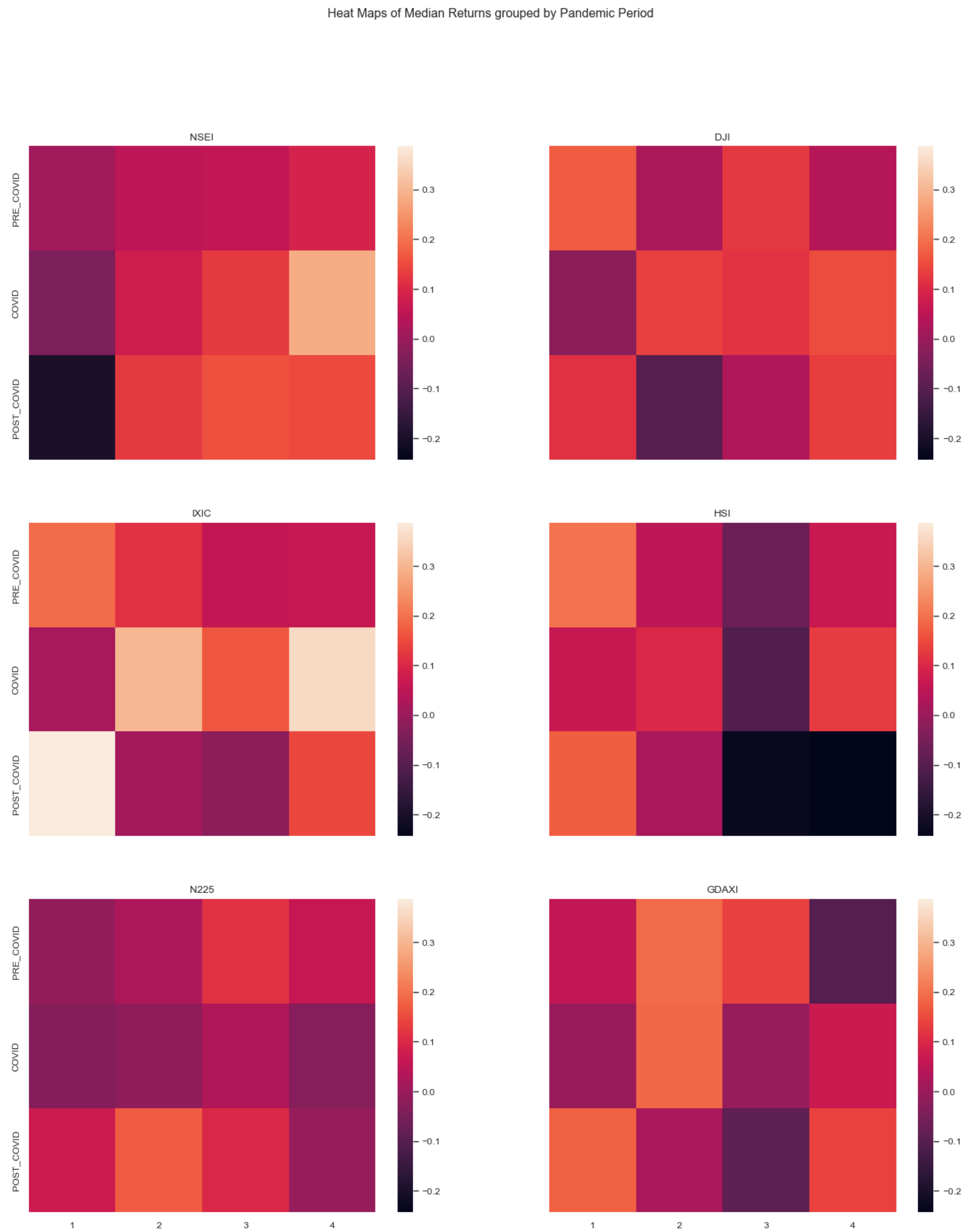
```
[ ]: #| label: heat-maps-mean-daily-returns-covid

performance_analytics_heat_maps(master, "PANDEMIC", "Heat Maps of Mean Returns␣
  ↪grouped by Pandemic Period", aggfunc = "mean")
```



Again, NSEI appears to be the most consistent of all indexes. All indexes have bad first quarters during the Pandemic, but improve post Covid.

Next we look at heatmaps by Pandemic period and quarter showing median returns:

```
[ ]: #| label: heat-maps-median-daily-returns-covid

performance_analytics_heat_maps(master, "PANDEMIC", "Heat Maps of Median␣
↪Returns grouped by Pandemic Period")
```



Median returns tells a similar story over the Pandemic period - HSI in particular appears to have had the worst recovery.

Next, we try to estimate the time taken for each of the indexes to return to the Pre Covid levels - my approach is to find how many days it takes for each index to reach a value greater than or equal to the Pre Covid mean returns value:

[ ]:
```
#| label: return-to-pre-covid-levels

for i in range(6):
    pre_covid  = master.loc[(master['PANDEMIC'] == 'PRE_COVID'),  [COLUMNS[i]]]
    post_covid = master.loc[(master['PANDEMIC'] == 'POST_COVID'), [COLUMNS[i]]]

    mean_pre   = pre_covid.values.mean()
    post_count = np.where(post_covid[COLUMNS[i]].ge(mean_pre).values ==␣
 ↪True)[0][0]
    post_date  = post_covid.index[post_covid[COLUMNS[i]].ge(mean_pre)][0].date()

    print(f"{INDICES[i].rjust(5)} returned to pre-covid levels (mean {mean_pre:␣
 ↪2.4f}) on {post_date} after {post_count} trading day(s)")
```

```
 NSEI returned to pre-covid levels (mean  0.0334) on 2022-05-16 after 7 trading
day(s)
  DJI returned to pre-covid levels (mean  0.0337) on 2022-05-13 after 6 trading
day(s)
 IXIC returned to pre-covid levels (mean  0.0619) on 2022-05-10 after 3 trading
day(s)
  HSI returned to pre-covid levels (mean -0.0126) on 2022-05-11 after 4 trading
day(s)
 N225 returned to pre-covid levels (mean  0.0001) on 2022-05-06 after 1 trading
day(s)
GDAXI returned to pre-covid levels (mean  0.0114) on 2022-05-10 after 3 trading
day(s)
```

Interestingly, N225 returned to it's Pre Covid level after just 1 day.

Lets define NSEI_OPEN_DIR as 1 if NSEI Open at time t > NSEI Close at time t - 1, and 0 otherwise:

[ ]:
```
master["NSEI_OPEN_DIR"] = np.where(master["NSEI_OPEN"] > master["NSEI_CLOSE"].
 ↪shift(), 1, 0)
```

Lets look at the percentages of NSEI_OPEN_DIR = 1 by year:

[ ]:
```
table1 = master.groupby("YEAR", observed = False)[["NSEI_OPEN_DIR"]].sum()
table2 = master.groupby("YEAR", observed = False)[["NSEI_OPEN_DIR"]].count()
table  = ((table1["NSEI_OPEN_DIR"] / table2["NSEI_OPEN_DIR"]) * 100).round(2)

print("\nNifty Fifty Daily Movement\n")
print(f"\n{table}\n")
```

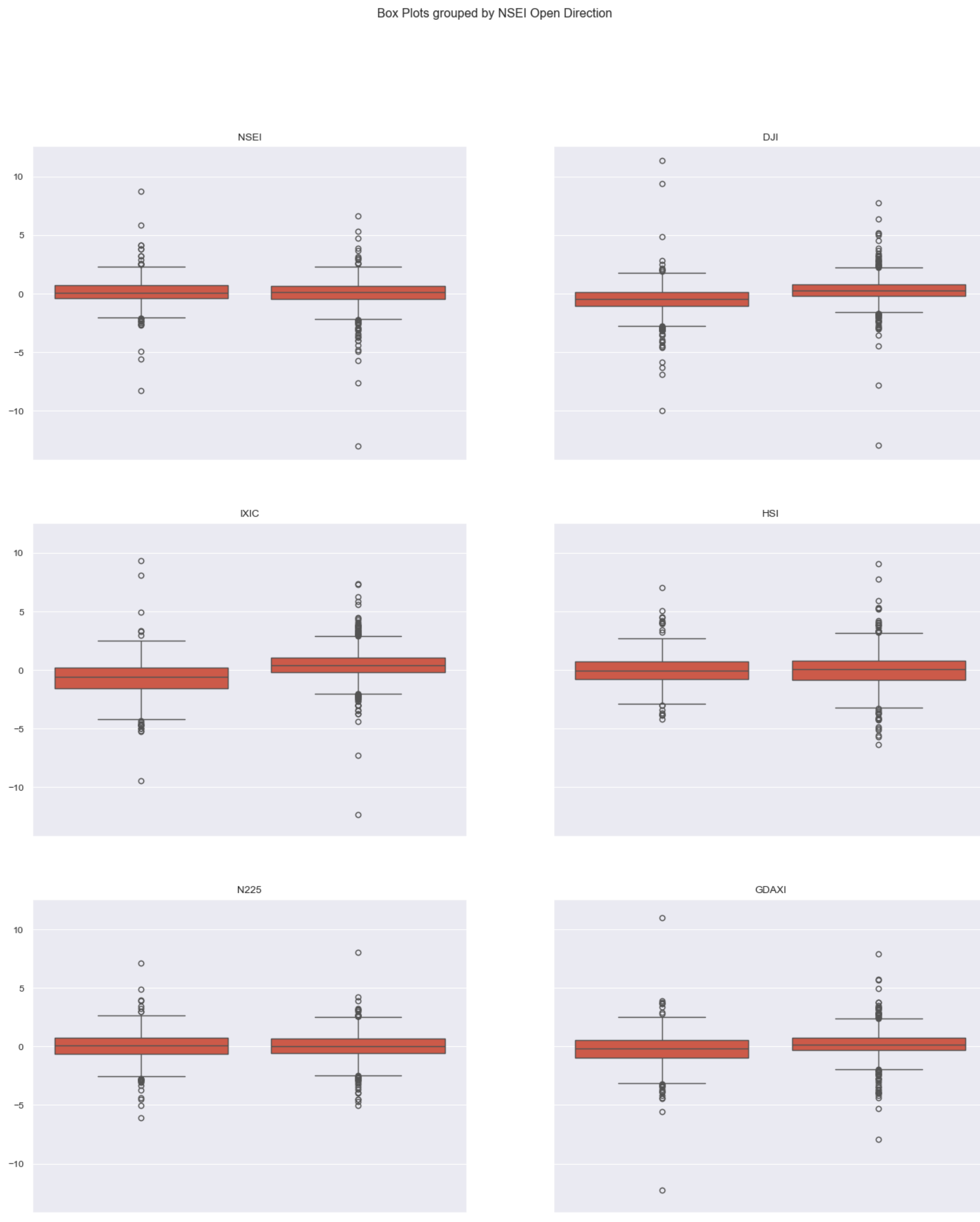
```

Nifty Fifty Daily Movement
```

```
YEAR
2018    70.38
2019    69.23
2020    70.61
2021    71.65
2022    59.23
2023    67.31
Name: NSEI_OPEN_DIR, dtype: float64
```

With the exception of 2022, every year has around 70% where NSEI_OPEN_DIR = 1.

Next we look at the indices for each category of NSEI_OPEN_DIR:

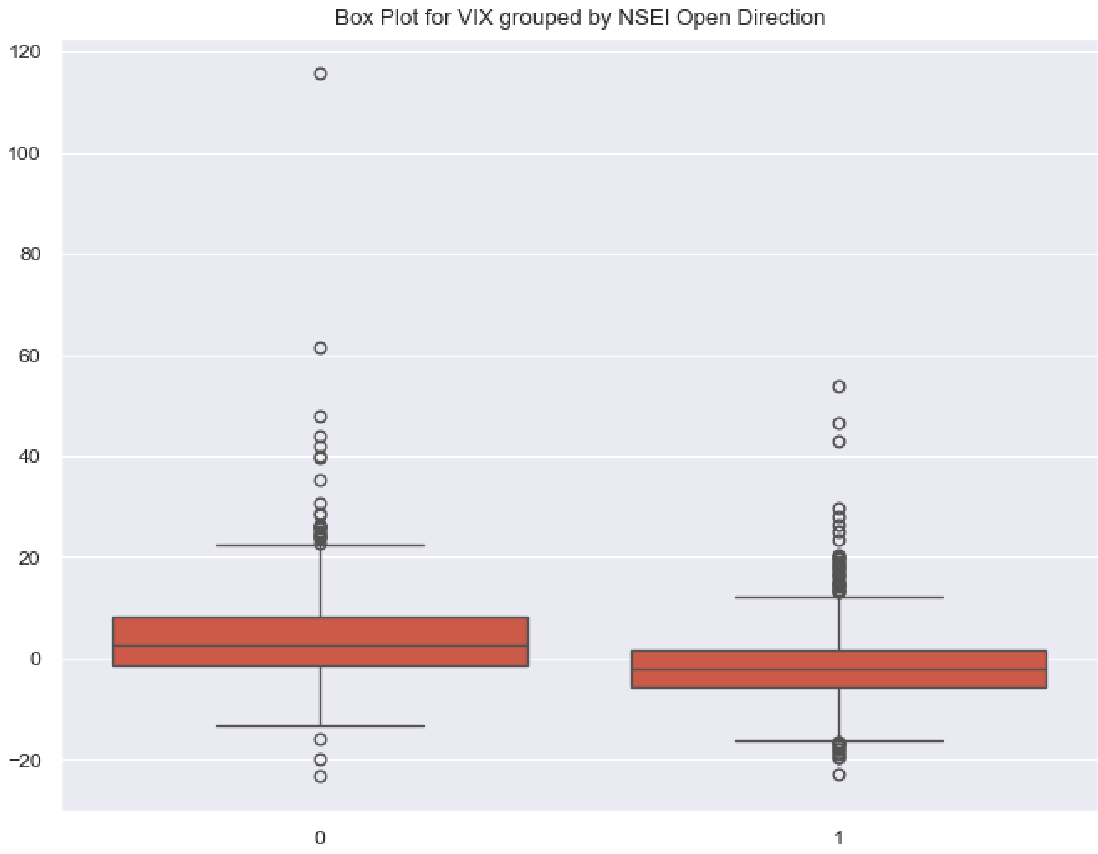
[ ]:
```
#| label: box-plots-daily-returns-grouped-by-open-dir

fig, axes = plt.subplots(3, 2, figsize = (16, 18), sharex = True, sharey = True)
fig.suptitle("Box Plots grouped by NSEI Open Direction")

for index in range(6):
    axes[index // 2, index % 2].set_title(INDICES[index])
    sns.boxplot(x = master["NSEI_OPEN_DIR"], y = master[COLUMNS[index]].
 ↪shift(), ax = axes[index // 2, index % 2])
    axes[index // 2, index % 2].set_xlabel("")
    axes[index // 2, index % 2].set_ylabel("")
```



we look at VIX separately - as it requires a different scale:

```
[ ]: #| label: box-plots-vix-daily-return-grouped-by-open-dir

plt.figure(figsize = (8, 6))
```

```
sns.boxplot(x = master["NSEI_OPEN_DIR"], y = master[COLUMNS[6]].shift())

plt.title(f"Box Plot for {INDICES[6]} grouped by NSEI Open Direction")
plt.xlabel("")
plt.ylabel("")

plt.show()
```



All of the box plots look consistent across each categroy of NSEI_OPEN_DIR, with the exceptions of IXIC and VIX.

1.3 Phase 3 - Training a Logistic Model

Before proceeding with modelling NSEI_OPEN_DIR, lets define, and add, some indicators and ratios:

[ ]:
```
RATIOS     = ["NSEI_HL_RATIO", "DJI_HL_RATIO"]
INDICATORS = ["NSEI_RSI", "DJI_RSI", "NSEI_TSI", "DJI_TSI"]
ALL_COLS   = COLUMNS + RATIOS + INDICATORS
```

Lets add NSEI_HL_RATIO and DJI_HL_RATIO:

[ ]:
```
master["NSEI_HL_RATIO"] = master["NSEI_HIGH"] / master["NSEI_LOW"]
master["DJI_HL_RATIO"]  = master["DJI_HIGH"]  / master["DJI_LOW"]
```

Lets add some technical indicators - RSI and TSI for NSEI and DJI:

[ ]:
```
master["NSEI_RSI"] = ta.momentum.rsi(master["NSEI_CLOSE"])
master["DJI_RSI"]  = ta.momentum.rsi(master["DJI_CLOSE"])

master["NSEI_TSI"] = ta.momentum.tsi(master["NSEI_CLOSE"])
master["DJI_TSI"]  = ta.momentum.tsi(master["DJI_CLOSE"])
```

Lets create a data frame containing all the data we will be working with:

[ ]:
```
data = pd.concat([master["NSEI_OPEN_DIR"].shift(-1), master[ALL_COLS]], axis =␣
 ↪1)
data.dropna(inplace = True)
data.head()
```

[ ]:
```
            NSEI_OPEN_DIR  NSEI_DAILY_RETURNS  DJI_DAILY_RETURNS  \
Date
2018-02-22            1.0           -0.141862           0.664177
2018-02-23            1.0            1.043559           1.392128
2018-02-26            1.0            0.872647           1.577556
2018-02-27            0.0           -0.267418          -1.163939
2018-02-28            0.0           -0.582229          -1.498739

            IXIC_DAILY_RETURNS  HSI_DAILY_RETURNS  N225_DAILY_RETURNS  \
Date
2018-02-22           -0.112772          -1.483242           -1.066738
2018-02-23            1.765585           0.973627            0.719252
2018-02-26            1.145773           0.740168            1.191496
2018-02-27           -1.227654          -0.729999            1.066320
2018-02-28           -0.782232          -1.355797           -1.436450

            GDAXI_DAILY_RETURNS  VIX_DAILY_RETURNS  NSEI_HL_RATIO  \
Date
2018-02-22            -0.068803          -6.493512       1.005502
2018-02-23             0.175574         -11.912391       1.009854
2018-02-26             0.346449          -4.184352       1.006915
2018-02-27            -0.289850          17.658227       1.008959
2018-02-28            -0.439373           6.777839       1.007069

            DJI_HL_RATIO   NSEI_RSI    DJI_RSI   NSEI_TSI   DJI_TSI
Date
2018-02-22      1.012146  35.462139  46.645122 -30.229045 -9.335285
2018-02-23      1.011394  43.991068  52.167111 -27.688141 -7.175461
```

```
2018-02-26      1.013160  50.003304  57.597238 -23.576140 -3.663260
2018-02-27      1.015449  48.278103  52.762870 -20.700940 -2.074670
2018-02-28      1.022129  44.673823  47.319430 -19.286991 -2.368269
```

Lets split our data into features (X) and labels (y):

```
[ ]: X = data[ALL_COLS]
     y = data['NSEI_OPEN_DIR']
```

And lets add an intercept to the features:

```
[ ]: X.insert(loc = 0, column = "Intercept", value = 1)
```

Now we spit our data into training and testing sets - with an 80 / 20 train / test split:

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
      ↪random_state = 1337)
```

We define a function that will prune any features that are either found to be insignificant, or that are found to be collinear:

```
[ ]: def prune(X, y, verbose = True):
         dropped = []
         while True:
             model = Logit(y, X).fit(disp = 0)

             insignificant = [p for p in zip(model.pvalues.index[1:], model.
      ↪pvalues[1:]) if p[1] > 0.05]

             values    = [variance_inflation_factor(model.model.exog, i) for i in␣
      ↪range(1, model.model.exog.shape[1])]
             colinear = [val for val in zip(model.model.exog_names[1:], values) if␣
      ↪val[1] > 5]

             if insignificant:
                 insignificant.sort(key = lambda p: -p[1])

                 if verbose:
                     print(f"dropping {insignificant[0][0]} with p-value␣
      ↪{insignificant[0][1]}")

                 X = X.drop([insignificant[0][0]], axis = 1)
                 dropped.append(insignificant[0][0])

             elif colinear:
                 colinear.sort(key = lambda c: -c[1])

                 if verbose:
                     print(f"dropping {colinear[0][0]} with vif {colinear[0][1]}")
```

```
            X = X.drop([colinear[0][0]], axis = 1)
            dropped.append(colinear[0][0])

        else:
            return model, dropped


model, dropped = prune(X_train, y_train)
```

```
dropping DJI_DAILY_RETURNS with p-value 0.7234766099770011
dropping GDAXI_DAILY_RETURNS with p-value 0.6162105670376612
dropping NSEI_HL_RATIO with p-value 0.4277618505298021
dropping DJI_HL_RATIO with p-value 0.1563055988923202
dropping NSEI_DAILY_RETURNS with p-value 0.13281329048460666
dropping NSEI_TSI with vif 5.865700460659149
dropping NSEI_RSI with p-value 0.7783762272653001
```

The function outputs a list of pruned features, together with the associated p-value or vif value. The function returns the pruned model, together with a list of pruned feature names.

Lets look at the summary of the model returned:

[ ]:
```
#| label: logistic-model-summary

model.summary()
```

[ ]:

Dep. Variable:	NSEI_OPEN_DIR	**No. Observations:**	1220
Model:	Logit	**Df Residuals:**	1213
Method:	MLE	**Df Model:**	6
Date:	Wed, 10 Jul 2024	**Pseudo R-squ.:**	0.1375
Time:	19:08:10	**Log-Likelihood:**	-660.02
converged:	True	**LL-Null:**	-765.23
Covariance Type:	nonrobust	**LLR p-value:**	1.141e-42

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
Intercept	-1.4041	0.656	-2.139	0.032	-2.690	-0.118
IXIC_DAILY_RETURNS	0.4552	0.075	6.093	0.000	0.309	0.602
HSI_DAILY_RETURNS	-0.1395	0.053	-2.632	0.008	-0.243	-0.036
N225_DAILY_RETURNS	-0.1960	0.068	-2.897	0.004	-0.329	-0.063
VIX_DAILY_RETURNS	-0.0397	0.013	-3.054	0.002	-0.065	-0.014
DJI_RSI	0.0447	0.013	3.415	0.001	0.019	0.070
DJI_TSI	-0.0205	0.008	-2.660	0.008	-0.036	-0.005

And now lets look at the list of dropped feattures:

[ ]:
```
#| label: logistic-model-dropped-features

print("\n".join(dropped))
```

```
DJI_DAILY_RETURNS
GDAXI_DAILY_RETURNS
NSEI_HL_RATIO
DJI_HL_RATIO
NSEI_DAILY_RETURNS
NSEI_TSI
NSEI_RSI
```

And lets also look at the variance inflation factors for each of the retained features:

[ ]:
```
#| label: logistic-model-vif

vif_data = pd.DataFrame()
vif_data["Feature"] = model.model.exog_names[1:]
vif_data["VIF"]     = [variance_inflation_factor(model.model.exog, i) for i in␣
 ↪range(1, model.model.exog.shape[1])]
vif_data
```

[ ]:

	Feature	VIF
0	IXIC_DAILY_RETURNS	2.073867
1	HSI_DAILY_RETURNS	1.244922
2	N225_DAILY_RETURNS	1.353286
3	VIX_DAILY_RETURNS	1.994009
4	DJI_RSI	4.850250
5	DJI_TSI	4.379409

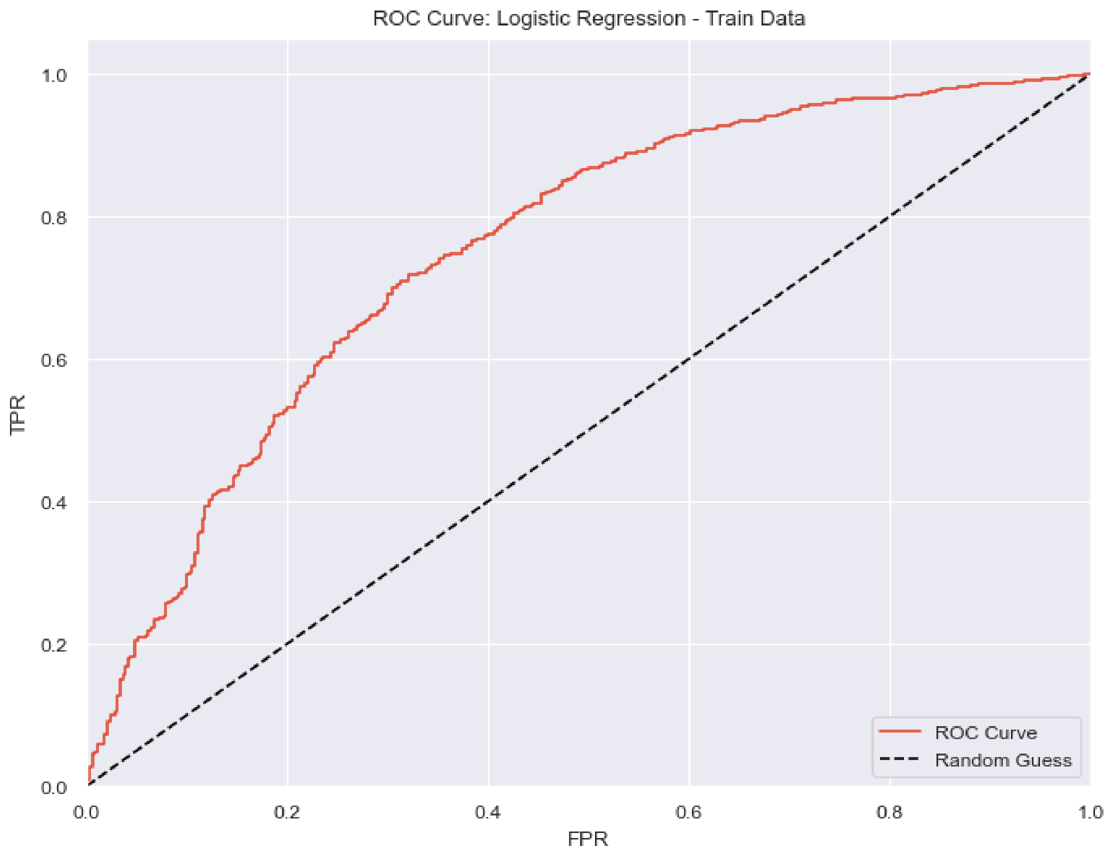
As we can see, all retained features have a vif of less than 5.

Let us define a function to plot ROC curves which we will use throughout this report:

[ ]:
```
def performance_analytics_roc_curve(fpr, tpr, title = "ROC Curve"):
    plt.figure(figsize = (8, 6))

    plt.plot(fpr, tpr, label = 'ROC Curve')
    plt.plot([0, 1], [0, 1], 'k--', label = 'Random Guess')

    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])

    plt.title(title)
    plt.xlabel('FPR')
    plt.ylabel('TPR')

    plt.legend(loc = 'lower right')
    plt.show()
```

Let us plot the ROC curve for the training data:

```
[ ]: #| label: logistic-model-roc-curve-train-data

y_pred = model.predict(X_train.drop(dropped, axis = 1))
fpr, tpr, thresholds = roc_curve(y_train, y_pred)

performance_analytics_roc_curve(fpr, tpr, title = "ROC Curve: Logistic␣
↪Regression - Train Data")
```



Let us find the AUC for the training data:

```
[ ]: #| label: logistic-model-auc-train-data

train_auc_roc = roc_auc_score(y_train, y_pred)
print(f'Train Data - AUC ROC: {train_auc_roc}')
```

```
Train Data - AUC ROC: 0.7529115595469844
```

Now we find the optimal threshold that balances sensitivity and specificity:

```
[ ]: optimal_threshold = round(thresholds[np.argmax(tpr - fpr)], 3)
print(f'Optimal Threshold: {optimal_threshold}')
```

```
Optimal Threshold: 0.684
```

Next we generate a classification report for the training data with the optimal threshold:

```
[ ]: #| label: logistic-model-classification-report-train-data

y_pred_class = np.where(y_pred <= optimal_threshold, 0, 1)
print(classification_report(y_train, y_pred_class))
```

```
              precision    recall  f1-score   support

         0.0       0.53      0.68      0.60       391
         1.0       0.83      0.72      0.77       829

    accuracy                           0.70      1220
   macro avg       0.68      0.70      0.68      1220
weighted avg       0.73      0.70      0.71      1220
```

Next we generate a confusion matrix for the training data with the optimal threshold:

```
[ ]: #| label: logistic-model-confusion-matrix-train-data

table_train = pd.crosstab(y_pred_class, y_train)
print(table_train)
```

```
NSEI_OPEN_DIR  0.0  1.0
row_0
0              265  234
1              126  595
```

And finally, we obtain the sensitivity and specificity for the training data:

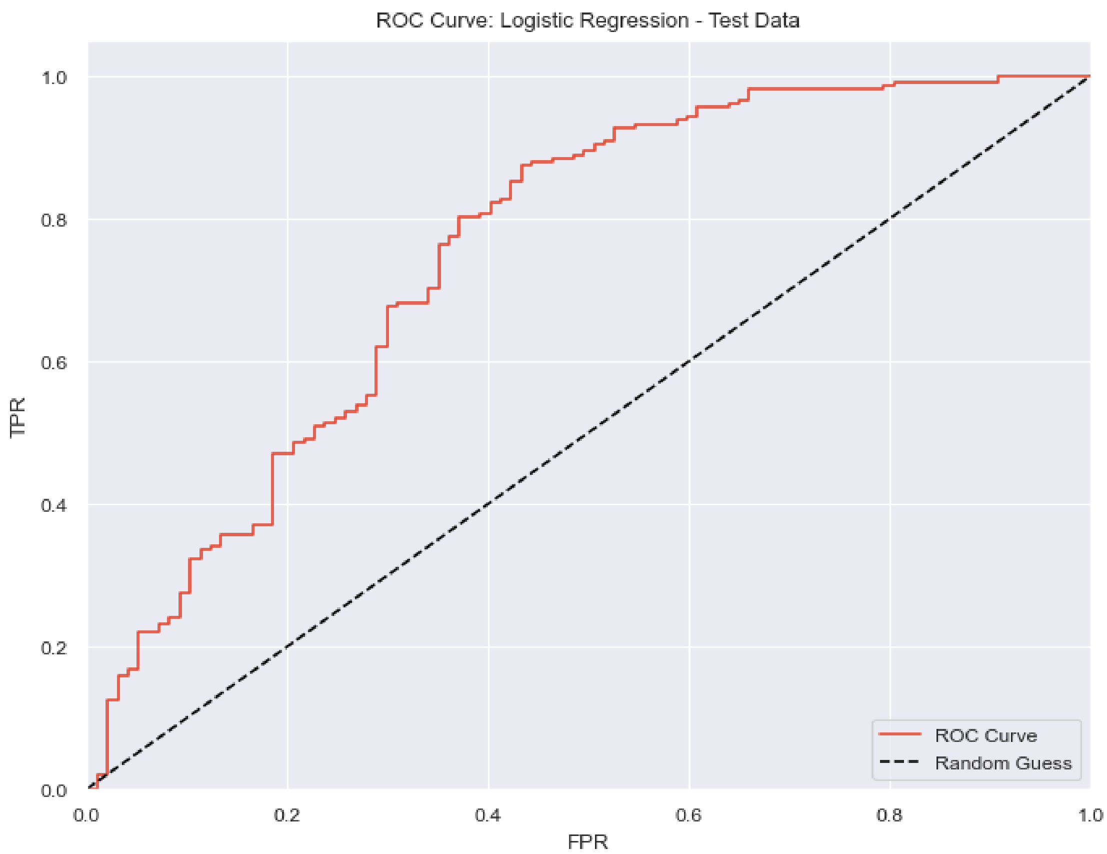
```
[ ]: #| label: logistic-model-sens-spec-train-data

sensitivity = round((table_train.iloc[1, 1] / (table_train.iloc[0, 1] +␣
 ↪table_train.iloc[1, 1])) * 100, 2)
specificity = round((table_train.iloc[0, 0] / (table_train.iloc[0, 0] +␣
 ↪table_train.iloc[1, 0])) * 100, 2)

print(f"Train Data - Sensitivity: {sensitivity}%")
print(f"Train Data - Specificity: {specificity}%")
```

```
Train Data - Sensitivity: 71.77%
Train Data - Specificity: 67.77%
```

Now we do the same for the test data - starting with obtaining the ROC curve:

```
[ ]: #| label: logistic-model-roc-curve-test-data

y_test_pred = model.predict(X_test.drop(dropped, axis = 1))
```

```
fpr, tpr, thresholds = roc_curve(y_test, y_test_pred)

performance_analytics_roc_curve(fpr, tpr, title = "ROC Curve: Logistic␣
  ↪Regression - Test Data")
```



Next, we obtain the AUC for the test data:

[ ]:
```
#| label: logistic-model-auc-test-data

test_auc_roc = roc_auc_score(y_test, y_test_pred)
print(f'Test Data - AUC ROC: {test_auc_roc}')
```

```
Test Data - AUC ROC: 0.7520816812053925
```

Next we generate a classification report for the test data (with the previously calculated optimal threshold):

[ ]:
```
#| label: logistic-model-classification-report-test-data

y_test_pred_class = np.where(y_test_pred <= optimal_threshold,  0, 1)
print(classification_report(y_test, y_test_pred_class))
```

```
              precision    recall  f1-score   support

         0.0       0.53      0.65      0.58        97
         1.0       0.82      0.73      0.77       208

    accuracy                           0.70       305
   macro avg       0.67      0.69      0.67       305
weighted avg       0.72      0.70      0.71       305
```

Next we generate a confusion matrix for the test data (with the previously calculated optimal threshold):

```
[ ]: #| label: logistic-model-confusion-matrix-test-data

table_test = pd.crosstab(y_test_pred_class, y_test)
print(table_test)
```

```
NSEI_OPEN_DIR  0.0  1.0
row_0
0               63   57
1               34  151
```

And finally, we obtain the sensitivity and specificity for the test data:

```
[ ]: #| label: logistic-model-sens-spec-test-data

sensitivity = round((table_test.iloc[1, 1] / (table_test.iloc[0, 1] +␣
 ↪table_test.iloc[1, 1])) * 100, 2)
specificity = round((table_test.iloc[0, 0] / (table_test.iloc[0, 0] +␣
 ↪table_test.iloc[1, 0])) * 100, 2)

print(f"Test Data - Sensitivity: {sensitivity}%")
print(f"Test Data - Specificity: {specificity}%")
```

```
Test Data - Sensitivity: 72.6%
Test Data - Specificity: 64.95%
```

Reviewing the AUC for both train and test data - we see they are very close implying consistency in the performance of the models:

```
[ ]: #| label: logistic-model-auc-comparison

print(f'Train Data - AUC ROC: {train_auc_roc}')
print(f' Test Data - AUC ROC: {test_auc_roc}')
```

```
Train Data - AUC ROC: 0.7529115595469844
 Test Data - AUC ROC: 0.7520816812053925
```

But the accuracy of the model is not great. We could obtain better results by selecting a different classification model.

1.4 Phase 4 - Compare Models

Next, we compare the performance of a number of different models. We first implement two functions that will help us:

```
[ ]: def model_metrics_plots(X_train, X_test, y_train, y_test, model, name =␣
     ↪"MODEL"):
         model.fit(X_train, y_train)

         y_train_pred_prob = model.predict_proba(X_train)
         train_fpr, train_tpr, _ = roc_curve(y_train, y_train_pred_prob[:, 1])

         y_test_pred_prob = model.predict_proba(X_test)
         test_fpr, test_tpr, _ = roc_curve(y_test, y_test_pred_prob[:, 1])

         fig, axes = plt.subplots(1, 2, figsize = (16, 6))
         fig.suptitle("ROC Curves")

         axes[0].set_title(f"{name} - Train Data")
         axes[0].plot(train_fpr, train_tpr, label = 'ROC Curve')
         axes[0].plot([0, 1], [0, 1], 'k--', label = 'Random Guess')
         axes[0].legend(loc = 'lower right')
         axes[0].set_xlabel("FPR")
         axes[0].set_ylabel("TPR")
         axes[0].set_xlim([0.0, 1.0])
         axes[0].set_ylim([0.0, 1.05])

         axes[1].set_title(f"{name} - Test Data")
         axes[1].plot(test_fpr, test_tpr, label = 'ROC Curve')
         axes[1].plot([0, 1], [0, 1], 'k--', label = 'Random Guess')
         axes[1].legend(loc = 'lower right')
         axes[1].set_xlabel("FPR")
         axes[1].set_ylabel("TPR")
         axes[1].set_xlim([0.0, 1.0])
         axes[1].set_ylim([0.0, 1.05])



     def model_metrics_data(X_train, X_test, y_train, y_test, model, name = "MODEL"):
         model.fit(X_train, y_train)

         y_train_pred_prob = model.predict_proba(X_train)
         train_fpr, train_tpr, thresholds = roc_curve(y_train, y_train_pred_prob[:,␣
     ↪1])

         y_test_pred_prob = model.predict_proba(X_test)

         optimal_threshold = round(thresholds[np.argmax(train_tpr - train_fpr)], 3)
```

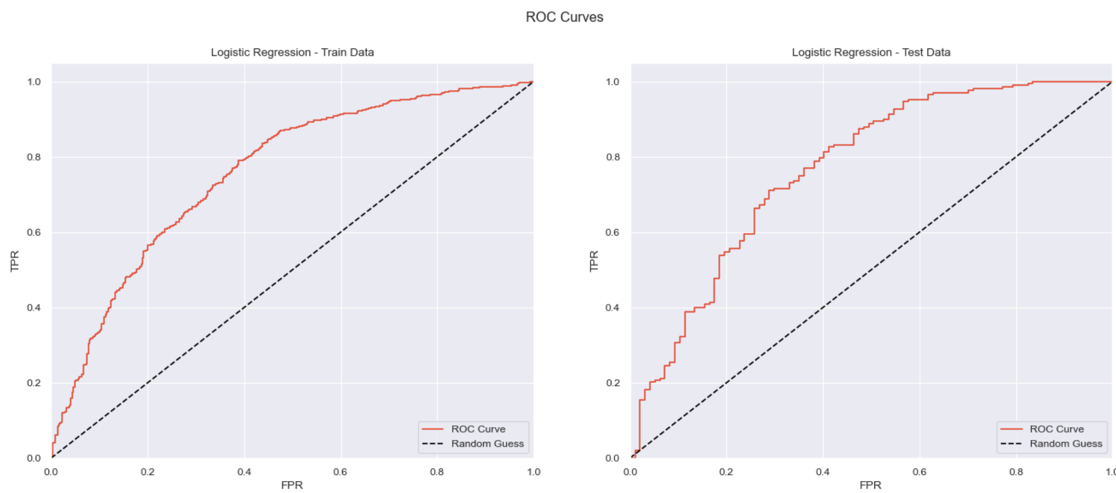
```python
    train_auc_roc = roc_auc_score(y_train, y_train_pred_prob[:, 1])
    y_train_pred_class = np.where(y_train_pred_prob[:, 1] <= optimal_threshold, 0, 1)
    train_table = pd.crosstab(y_train_pred_class, y_train)
    train_sensitivity = round((train_table.iloc[1, 1] / (train_table.iloc[0, 1] + train_table.iloc[1, 1])) * 100, 2)
    train_specificity = round((train_table.iloc[0, 0] / (train_table.iloc[0, 0] + train_table.iloc[1, 0])) * 100, 2)

    test_auc_roc = roc_auc_score(y_test, y_test_pred_prob[:, 1])
    y_test_pred_class = np.where(y_test_pred_prob[:, 1] <= optimal_threshold, 0, 1)
    test_table = pd.crosstab(y_test_pred_class, y_test)
    test_sensitivity = round((test_table.iloc[1, 1] / (test_table.iloc[0, 1] + test_table.iloc[1, 1])) * 100, 2)
    test_specificity = round((test_table.iloc[0, 0] / (test_table.iloc[0, 0] + test_table.iloc[1, 0])) * 100, 2)

    print()
    print(f"{name}")
    print()

    print(f"\nTrain Data - Classification Report:\n")
    print(classification_report(y_train, y_train_pred_class))
    print(f"\nTrain Data - Confusion Matrix:\n\n{train_table}\n")

    print(f"\n Test Data - Classification Report:\n")
    print(classification_report(y_test, y_test_pred_class))
    print(f"\n Test Data - Confusion Matrix:\n\n{test_table}\n")

    print()

    print(f"Train Data - Sensitivity for cut-off {optimal_threshold}: {train_sensitivity}%")
    print(f" Test Data - Sensitivity for cut-off {optimal_threshold}: {test_sensitivity}%\n")

    print(f"Train Data - Specificity for cut-off {optimal_threshold}: {train_specificity}%")
    print(f" Test Data - Specificity for cut-off {optimal_threshold}: {test_specificity}%\n")

    print(f"Train Data - AUC ROC: {train_auc_roc}")
    print(f" Test Data - AUC ROC: {test_auc_roc}\n")
```

We prepare our data:

[ ]:
```
X = data[ALL_COLS].values
y = data['NSEI_OPEN_DIR'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 1337)
```

We once again look at a Logistic model, which will become our baseline:

[ ]:
```
#| label: model-metrics-plots-logistic

model_log = LogisticRegression(max_iter = 1000, random_state = 1337)
model_metrics_plots(X_train, X_test, y_train, y_test, model_log, name =␣
↪"Logistic Regression")
```



and the corresponding data:

[ ]:
```
#| label: model-metrics-data-logistic

model_metrics_data(X_train, X_test, y_train, y_test, model_log, name =␣
↪"Logistic Regression")
```

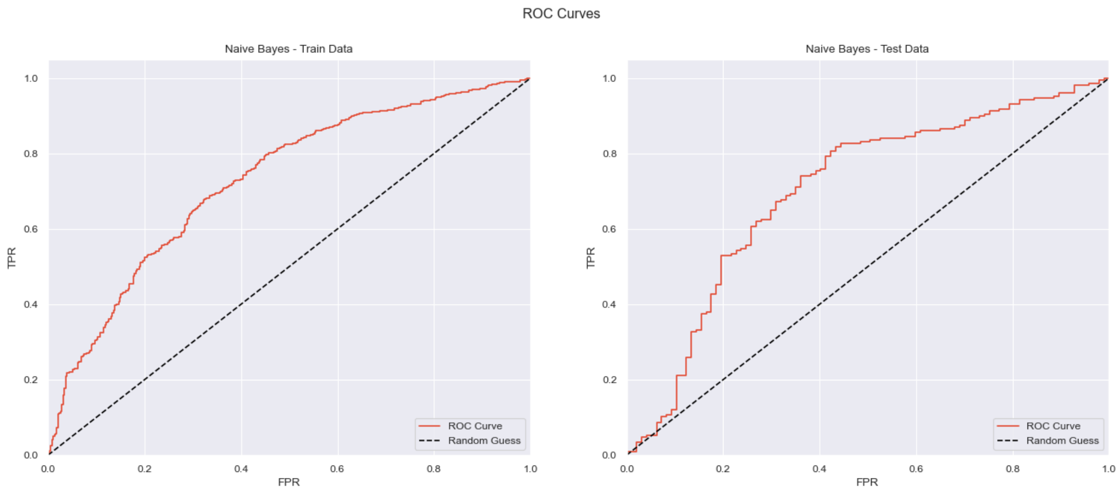
```
Logistic Regression


Train Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.58      0.61      0.59       391
         1.0       0.81      0.79      0.80       829
```

```
    accuracy                           0.73      1220
   macro avg       0.70      0.70      0.70      1220
weighted avg       0.74      0.73      0.73      1220


Train Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0      239  174
1      152  655


 Test Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.56      0.63      0.59        97
         1.0       0.82      0.77      0.79       208

    accuracy                           0.72       305
   macro avg       0.69      0.70      0.69       305
weighted avg       0.73      0.72      0.73       305


 Test Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0       61   48
1       36  160


Train Data - Sensitivity for cut-off 0.654: 79.01%
 Test Data - Sensitivity for cut-off 0.654: 76.92%

Train Data - Specificity for cut-off 0.654: 61.13%
 Test Data - Specificity for cut-off 0.654: 62.89%

Train Data - AUC ROC: 0.7584554774340638
 Test Data - AUC ROC: 0.767248215701824
```

We notice that the scikit-learn LogisticRegression model slightly outperforms the Statsmodels Logit model. We use the Statsmodels Logit model when we need to perform analysis of features. Moving to scikit-learn's LogisticRegression model after the model has been finalised is acceptable.

Next we look at Naive Bayes:

```
[ ]: #| label: model-metrics-plots-naive-bayes

model_nb = GaussianNB()
model_metrics_plots(X_train, X_test, y_train, y_test, model_nb, name = "Naive␣
↪Bayes")
```



and the corresponding data:

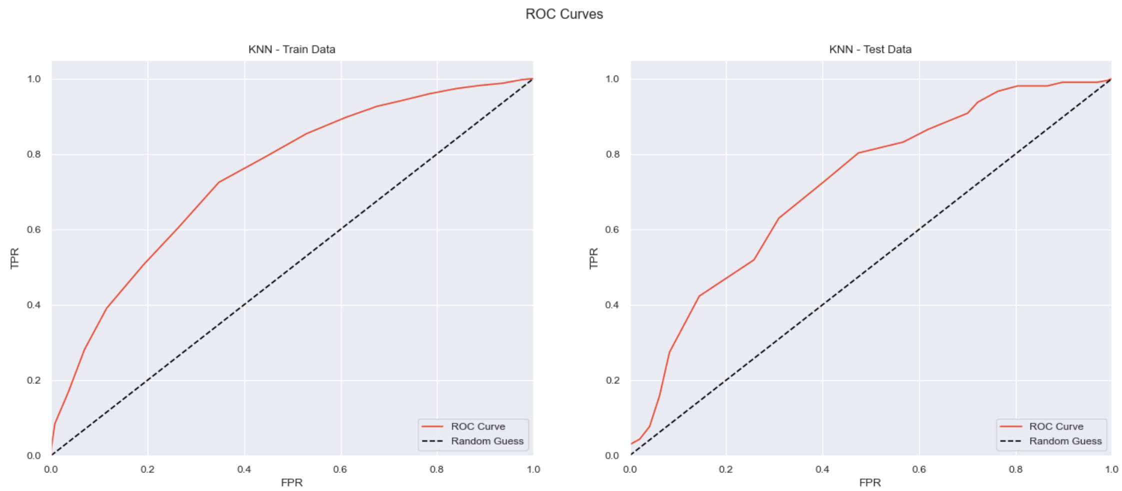
```
[ ]: #| label: model-metrics-data-naive-bayes

model_metrics_data(X_train, X_test, y_train, y_test, model_nb, name = "Naive␣
↪Bayes")
```

```
Naive Bayes


Train Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.50      0.68      0.57       391
         1.0       0.82      0.68      0.74       829

    accuracy                           0.68      1220
   macro avg       0.66      0.68      0.66      1220
weighted avg       0.71      0.68      0.69      1220


Train Data - Confusion Matrix:
```

```
col_0  0.0  1.0
row_0
0      264  266
1      127  563


 Test Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.47      0.70      0.56        97
         1.0       0.82      0.62      0.71       208

    accuracy                           0.65       305
   macro avg       0.64      0.66      0.63       305
weighted avg       0.71      0.65      0.66       305


 Test Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0       68   78
1       29  130


Train Data - Sensitivity for cut-off 0.916: 67.91%
 Test Data - Sensitivity for cut-off 0.916: 62.5%

Train Data - Specificity for cut-off 0.916: 67.52%
 Test Data - Specificity for cut-off 0.916: 70.1%

Train Data - AUC ROC: 0.7282400451658085
 Test Data - AUC ROC: 0.7022204599524188
```

The Naive Bayes model clearly performs worse than the Logistic model. Next, we look at a KNN classifier:

[ ]:
```
#| label: model-metrics-plots-knn

scaler = StandardScaler()
scaler.fit(X_train)

model_knn = KNeighborsClassifier(leaf_size = 10, n_neighbors = 30)
model_metrics_plots(X_train, X_test, y_train, y_test, model_knn, name = "KNN")
```



and the corresponding data:

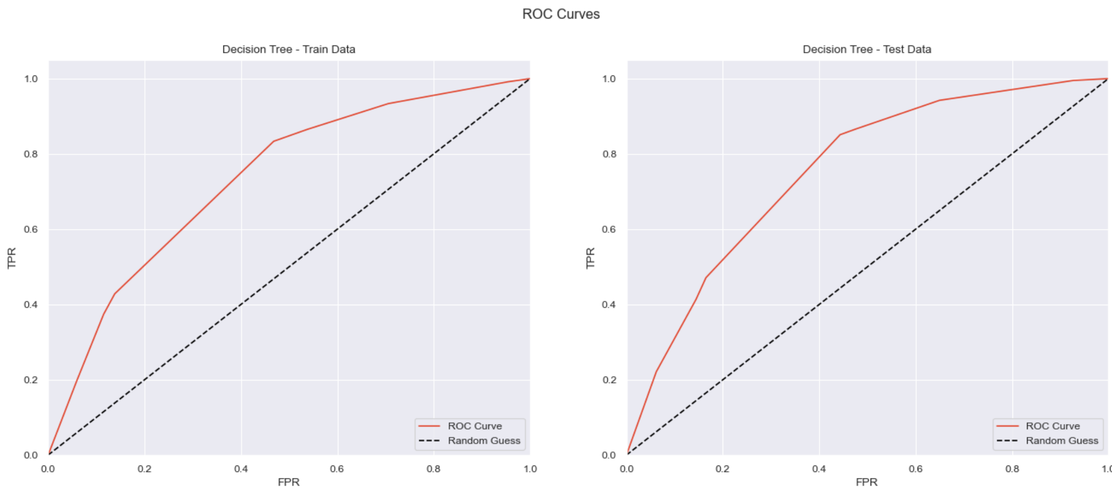
```
[ ]: #| label: model-metrics-data-knn

model_metrics_data(X_train, X_test, y_train, y_test, model_knn, name = "KNN")
```

```
KNN


Train Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.47      0.74      0.57       391
         1.0       0.83      0.60      0.70       829

    accuracy                           0.65      1220
   macro avg       0.65      0.67      0.64      1220
weighted avg       0.71      0.65      0.66      1220


Train Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0      288  328
1      103  501


 Test Data - Classification Report:
```

```
              precision    recall  f1-score   support

         0.0       0.47      0.69      0.56        97
         1.0       0.81      0.63      0.71       208

    accuracy                           0.65       305
   macro avg       0.64      0.66      0.63       305
weighted avg       0.70      0.65      0.66       305


 Test Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0       67   77
1       30  131


Train Data - Sensitivity for cut-off 0.7: 60.43%
 Test Data - Sensitivity for cut-off 0.7: 62.98%

Train Data - Specificity for cut-off 0.7: 73.66%
 Test Data - Specificity for cut-off 0.7: 69.07%

Train Data - AUC ROC: 0.7450939257540746
 Test Data - AUC ROC: 0.7160735527359239
```

The KNN model also performs badly when compared to the Logistic model. Next, we look at a Decision Tree classifier:

```
[ ]: #| label: model-metrics-plots-decision-tree

model_dt = DecisionTreeClassifier(max_depth = 10, min_samples_split = 0.4,␣
 ↪splitter = "random", random_state = 1337)
model_metrics_plots(X_train, X_test, y_train, y_test, model_dt, name =␣
 ↪"Decision Tree")
```



and the data:

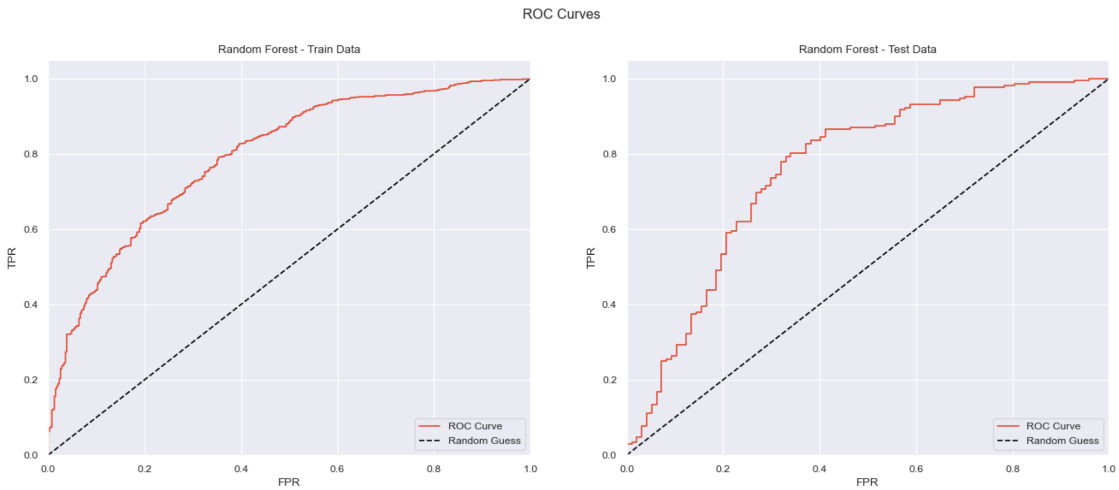
```
[ ]: #| label: model-metrics-data-decision-tree

model_metrics_data(X_train, X_test, y_train, y_test, model_dt, name = "Decision␣
↪Tree")
```

```
Decision Tree


Train Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.42      0.86      0.56       391
         1.0       0.87      0.43      0.57       829

    accuracy                           0.57      1220
   macro avg       0.64      0.65      0.57      1220
weighted avg       0.72      0.57      0.57      1220


Train Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0      337  474
1       54  355
```

```
 Test Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.42      0.84      0.56        97
         1.0       0.86      0.47      0.61       208

    accuracy                           0.59       305
   macro avg       0.64      0.65      0.59       305
weighted avg       0.72      0.59      0.59       305


 Test Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0       81  110
1       16   98


Train Data - Sensitivity for cut-off 0.723: 42.82%
 Test Data - Sensitivity for cut-off 0.723: 47.12%

Train Data - Specificity for cut-off 0.723: 86.19%
 Test Data - Specificity for cut-off 0.723: 83.51%

Train Data - AUC ROC: 0.7342235892626312
 Test Data - AUC ROC: 0.7527260111022998
```

The Decision Tree classifier performs very badly - with an accuracy of around 55%. Next, we look at a Random Forest classifier:

[ ]:
```
#| label: model-metrics-plots-random-forest

model_rf = RandomForestClassifier(max_depth = 10, min_samples_split = 0.2,␣
  ↪random_state = 1337)
model_metrics_plots(X_train, X_test, y_train, y_test, model_rf, name = "Random␣
  ↪Forest")
```



and the data:

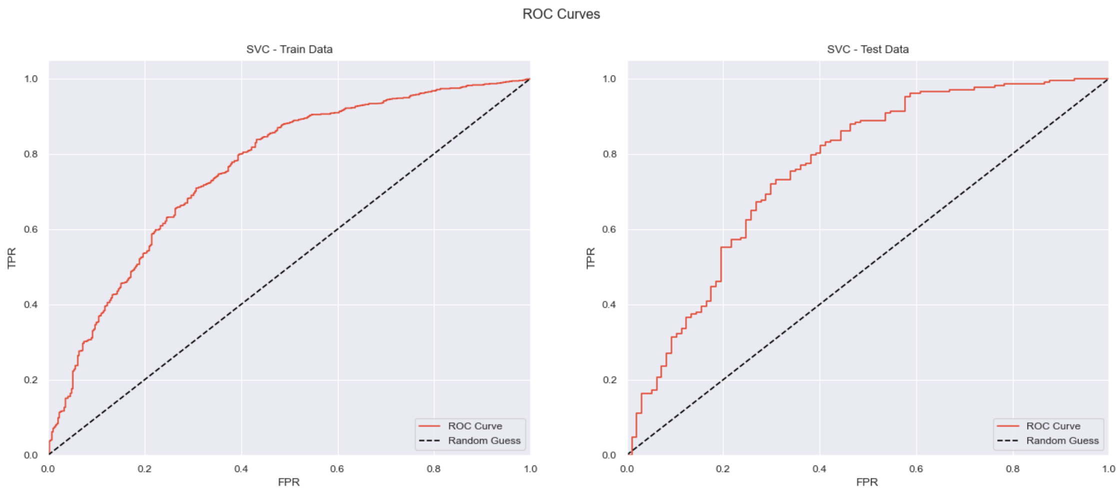
```
[ ]: #| label: model-metrics-data-random-forest

model_metrics_data(X_train, X_test, y_train, y_test, model_rf, name = "Random␣
 ↪Forest")
```

```
Random Forest


Train Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.59      0.65      0.62       391
         1.0       0.83      0.79      0.81       829

    accuracy                           0.74      1220
   macro avg       0.71      0.72      0.71      1220
weighted avg       0.75      0.74      0.75      1220


Train Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0      253  175
1      138  654
```

```
 Test Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.60      0.67      0.63        97
         1.0       0.84      0.79      0.81       208

    accuracy                           0.75       305
   macro avg       0.72      0.73      0.72       305
weighted avg       0.76      0.75      0.76       305


 Test Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0       65   43
1       32  165


Train Data - Sensitivity for cut-off 0.638: 78.89%
 Test Data - Sensitivity for cut-off 0.638: 79.33%

Train Data - Specificity for cut-off 0.638: 64.71%
 Test Data - Specificity for cut-off 0.638: 67.01%

Train Data - AUC ROC: 0.7938461585924558
 Test Data - AUC ROC: 0.7607057890563046
```

The Random Forest classifier performs reasonably well, but not as well as the baseline Logistic model. Next, we look at Support Vector Machine classifier:

[ ]:
```
#| label: model-metrics-plots-svm

model_svm = SVC(C = 1, kernel = "linear", probability = True, random_state =␣
↪1337)
model_metrics_plots(X_train, X_test, y_train, y_test, model_svm, name = "SVC")
```



and the data:

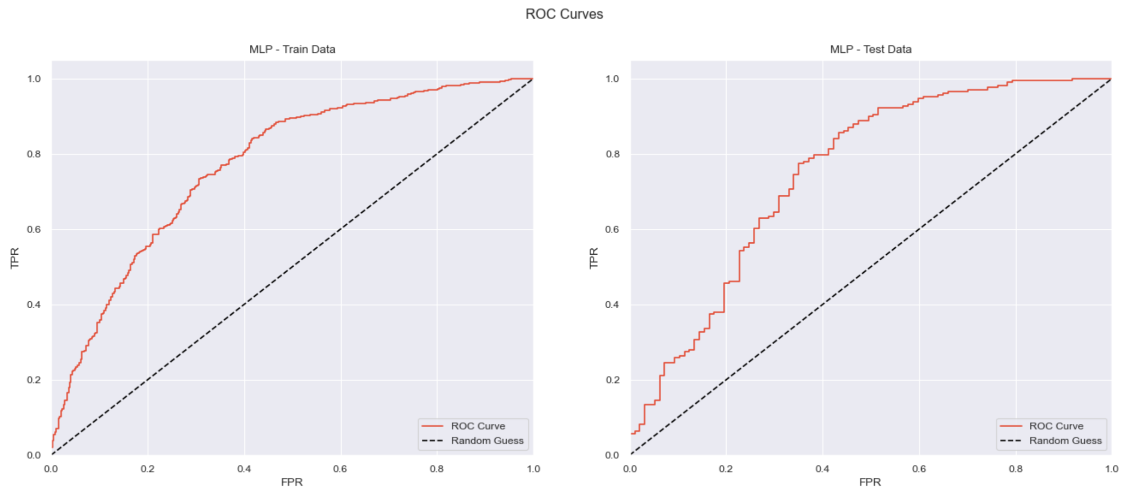
```
[ ]: #| label: model-metrics-data-svm

model_metrics_data(X_train, X_test, y_train, y_test, model_svm, name = "SVC")
```

```
SVC


Train Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.62      0.56      0.59       391
         1.0       0.80      0.84      0.82       829

    accuracy                           0.75      1220
   macro avg       0.71      0.70      0.70      1220
weighted avg       0.74      0.75      0.75      1220


Train Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0      219  133
1      172  696


 Test Data - Classification Report:
```

```
              precision    recall  f1-score   support

         0.0       0.61      0.59      0.60        97
         1.0       0.81      0.82      0.82       208

    accuracy                           0.75       305
   macro avg       0.71      0.70      0.71       305
weighted avg       0.75      0.75      0.75       305


 Test Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0       57   37
1       40  171


Train Data - Sensitivity for cut-off 0.642: 83.96%
 Test Data - Sensitivity for cut-off 0.642: 82.21%

Train Data - Specificity for cut-off 0.642: 56.01%
 Test Data - Specificity for cut-off 0.642: 58.76%

Train Data - AUC ROC: 0.759672547888406
 Test Data - AUC ROC: 0.763332672482157
```

This is the best performing model yet with a consistent accuracy of 75% and consistent AUC of ~0.76. Next, we move on to Multilayer Perceptron classifier:

[ ]:
```
#| label: model-metrics-plots-mlp

scaler = MinMaxScaler()
scaler.fit(X_train)

model_mlp = MLPClassifier(alpha = 0.001, max_iter = 1000, random_state = 1337)
model_metrics_plots(scaler.transform(X_train), scaler.transform(X_test),␣
 ↪y_train, y_test, model_mlp, name = "MLP")
```



and the data:

[ ]:
```
#| label: model-metrics-data-mlp

model_metrics_data(scaler.transform(X_train), scaler.transform(X_test),␣
↪y_train, y_test, model_mlp, name = "MLP")
```

```
MLP


Train Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.55      0.69      0.61       391
         1.0       0.84      0.73      0.78       829

    accuracy                           0.72      1220
   macro avg       0.69      0.71      0.70      1220
weighted avg       0.74      0.72      0.73      1220


Train Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0      271  221
1      120  608
```

```
 Test Data - Classification Report:

              precision    recall  f1-score   support

         0.0       0.52      0.66      0.58        97
         1.0       0.82      0.71      0.76       208

    accuracy                           0.70       305
   macro avg       0.67      0.69      0.67       305
weighted avg       0.72      0.70      0.70       305


 Test Data - Confusion Matrix:

col_0  0.0  1.0
row_0
0       64   60
1       33  148


Train Data - Sensitivity for cut-off 0.712: 73.34%
 Test Data - Sensitivity for cut-off 0.712: 71.15%

Train Data - Specificity for cut-off 0.712: 69.31%
 Test Data - Specificity for cut-off 0.712: 65.98%

Train Data - AUC ROC: 0.7711321377557159
 Test Data - AUC ROC: 0.7492069785884219
```

The Multilayer Perceptron model does reasonably well, but not as well as the SVM classifier. Overall, the SVM classifier performs the best.

Lets see if we can imporove on the SVM classifier with a deep learning model implemented using PyTorch. First, lets check to see if cuda is available:

[ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

Next we create a deep learning model:

[ ]:
```
class DL(nn.Module):

    def __init__(self, input_dim, hidden_dim, output_dim):
        super(DL, self).__init__()

        self.input  = nn.Linear(input_dim, hidden_dim)
```

```
        self.act_in = nn.ReLU()

        self.hidden1 = nn.Linear(hidden_dim, hidden_dim)
        self.act1 = nn.ReLU()
        self.hidden2 = nn.Linear(hidden_dim, hidden_dim)
        self.act2 = nn.ReLU()
        self.hidden3 = nn.Linear(hidden_dim, hidden_dim)
        self.act3 = nn.ReLU()
        self.hidden4 = nn.Linear(hidden_dim, hidden_dim)
        self.act4 = nn.ReLU()
        self.hidden5 = nn.Linear(hidden_dim, hidden_dim)
        self.act5 = nn.ReLU()
        self.hidden6 = nn.Linear(hidden_dim, hidden_dim)
        self.act6 = nn.ReLU()
        self.hidden7 = nn.Linear(hidden_dim, hidden_dim)
        self.act7 = nn.ReLU()

        self.output  = nn.Linear(hidden_dim, output_dim)
        self.act_out = nn.Sigmoid()

    def forward(self, x):
        x = self.act_in(self.input(x))

        x = self.act1(self.hidden1(x))
        x = self.act2(self.hidden2(x))
        x = self.act3(self.hidden3(x))
        x = self.act4(self.hidden4(x))
        x = self.act5(self.hidden5(x))
        x = self.act6(self.hidden6(x))
        x = self.act7(self.hidden7(x))

        return self.act_out(self.output(x))
```

Next we scale the data, and create a train test split:

```
[ ]: scaler = MinMaxScaler(feature_range = (0, 1))

X = scaler.fit_transform(data[ALL_COLS].values)
y = data['NSEI_OPEN_DIR'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 1337)
```

Next we convert the data into tensors, and move them onto the cuda device if available:

```
[ ]: X_train = torch.from_numpy(X_train).type(torch.Tensor)
X_test  = torch.from_numpy(X_test).type(torch.Tensor)
y_train = torch.from_numpy(y_train).type(torch.Tensor)
```

```
if torch.cuda.is_available():
    X_train = X_train.cuda()
    X_test  = X_test.cuda()
    y_train = y_train.cuda()
```

Next define our model parameters, create an instance of our model, and define our loss criterion and optimiser:

```
[ ]: input_dim  = X.shape[1]
hidden_dim = 64
output_dim = 1

model      = DL(input_dim, hidden_dim, output_dim)
model      = model.to(device)

criterion = nn.MSELoss()
optimiser = torch.optim.Adam(model.parameters(), lr = 0.0001)
```

Next we create functions to train our model, with and without batches, and make use of it:

```
[ ]: def train(X, y, model, criterion, optimiser, epochs = 500):
    losses = []

    for epoch in range(epochs):
        out  = model(X)
        loss = criterion(out, y)

        losses.append(loss.item())

        optimiser.zero_grad()
        loss.backward()
        optimiser.step()

        if epoch % 10 == 9:
            print(f"Epoch {epoch + 1:>3} - MSE: {loss.item()}")

    return losses

def train_batched(X, y, model, criterion, optimiser, epochs = 500, batch_size =␣
 ↪20, shuffle = False):
    losses = []

    dataset    = utils.TensorDataset(X, y)
    dataloader = utils.DataLoader(dataset, batch_size = batch_size, shuffle =␣
 ↪shuffle)

    for epoch in range(epochs):
```

```python
        for batch, (X_batch, y_batch) in enumerate(dataloader):
            out  = model(X_batch)
            loss = criterion(out, y_batch)

            losses.append(loss.item())

            optimiser.zero_grad()
            loss.backward()
            optimiser.step()

            if batch % 10 == 9:
                print(f"Epoch {epoch + 1:>3} - Batch {batch + 1:>3} - MSE:␣
↪{loss.item()}")

    return losses

# losses = train(X_train, y_train[:, None], model, criterion, optimiser)
losses = train_batched(X_train, y_train[:, None], model, criterion, optimiser,␣
↪batch_size = 100, shuffle = True)
```

```
Epoch   1 - Batch  10 - MSE: 0.24581755697727203
Epoch   2 - Batch  10 - MSE: 0.2428935021162033
Epoch   3 - Batch  10 - MSE: 0.24090971052646637
Epoch   4 - Batch  10 - MSE: 0.2452872395515442
Epoch   5 - Batch  10 - MSE: 0.24358420073986053
Epoch   6 - Batch  10 - MSE: 0.2402580976486206
Epoch   7 - Batch  10 - MSE: 0.23894134163856506
Epoch   8 - Batch  10 - MSE: 0.23425158858299255
Epoch   9 - Batch  10 - MSE: 0.22734546661376953
Epoch  10 - Batch  10 - MSE: 0.24289806187152863
Epoch  11 - Batch  10 - MSE: 0.22648021578788757
Epoch  12 - Batch  10 - MSE: 0.2319982498884201
Epoch  13 - Batch  10 - MSE: 0.221490740776062
Epoch  14 - Batch  10 - MSE: 0.2225533127784729
Epoch  15 - Batch  10 - MSE: 0.2453363686800003
Epoch  16 - Batch  10 - MSE: 0.22490368783473969
Epoch  17 - Batch  10 - MSE: 0.22733283042907715
Epoch  18 - Batch  10 - MSE: 0.21217261254787445
Epoch  19 - Batch  10 - MSE: 0.20337630808353424
Epoch  20 - Batch  10 - MSE: 0.2104637324810028
Epoch  21 - Batch  10 - MSE: 0.22966529428958893
Epoch  22 - Batch  10 - MSE: 0.21050739288330078
Epoch  23 - Batch  10 - MSE: 0.2088465839624405
Epoch  24 - Batch  10 - MSE: 0.22618086636066437
Epoch  25 - Batch  10 - MSE: 0.2149863839149475
Epoch  26 - Batch  10 - MSE: 0.22453854978084564
Epoch  27 - Batch  10 - MSE: 0.1945880502462387
Epoch  28 - Batch  10 - MSE: 0.20430682599544525
```

```
Epoch  29 - Batch  10 - MSE: 0.2090795785188675
Epoch  30 - Batch  10 - MSE: 0.23998676240444183
Epoch  31 - Batch  10 - MSE: 0.23788504302501678
Epoch  32 - Batch  10 - MSE: 0.21982870995998383
Epoch  33 - Batch  10 - MSE: 0.22588509321212769
Epoch  34 - Batch  10 - MSE: 0.23842665553092957
Epoch  35 - Batch  10 - MSE: 0.19864657521247864
Epoch  36 - Batch  10 - MSE: 0.2045675367116928
Epoch  37 - Batch  10 - MSE: 0.20656293630599976
Epoch  38 - Batch  10 - MSE: 0.21950800716876984
Epoch  39 - Batch  10 - MSE: 0.2241199016571045
Epoch  40 - Batch  10 - MSE: 0.18328139185905457
Epoch  41 - Batch  10 - MSE: 0.19495323300361633
Epoch  42 - Batch  10 - MSE: 0.2197721302509308
Epoch  43 - Batch  10 - MSE: 0.2022537887096405
Epoch  44 - Batch  10 - MSE: 0.21757613122463226
Epoch  45 - Batch  10 - MSE: 0.21592575311660767
Epoch  46 - Batch  10 - MSE: 0.18501685559749603
Epoch  47 - Batch  10 - MSE: 0.2049361616373062
Epoch  48 - Batch  10 - MSE: 0.17691770195960999
Epoch  49 - Batch  10 - MSE: 0.19991381466388702
Epoch  50 - Batch  10 - MSE: 0.19326888024806976
Epoch  51 - Batch  10 - MSE: 0.23071834444999695
Epoch  52 - Batch  10 - MSE: 0.2128244787454605
Epoch  53 - Batch  10 - MSE: 0.19277410209178925
Epoch  54 - Batch  10 - MSE: 0.20953021943569183
Epoch  55 - Batch  10 - MSE: 0.19199645519256592
Epoch  56 - Batch  10 - MSE: 0.1940612494945526
Epoch  57 - Batch  10 - MSE: 0.1877998262643814
Epoch  58 - Batch  10 - MSE: 0.19676573574543
Epoch  59 - Batch  10 - MSE: 0.21654623746871948
Epoch  60 - Batch  10 - MSE: 0.16983123123645782
Epoch  61 - Batch  10 - MSE: 0.17954587936401367
Epoch  62 - Batch  10 - MSE: 0.22203460335731506
Epoch  63 - Batch  10 - MSE: 0.17913733422756195
Epoch  64 - Batch  10 - MSE: 0.15494060516357422
Epoch  65 - Batch  10 - MSE: 0.1639593094587326
Epoch  66 - Batch  10 - MSE: 0.18090836703777313
Epoch  67 - Batch  10 - MSE: 0.17307689785957336
Epoch  68 - Batch  10 - MSE: 0.17274436354637146
Epoch  69 - Batch  10 - MSE: 0.13173899054527283
Epoch  70 - Batch  10 - MSE: 0.1719854772090912
Epoch  71 - Batch  10 - MSE: 0.17228080332279205
Epoch  72 - Batch  10 - MSE: 0.197896346449852
Epoch  73 - Batch  10 - MSE: 0.15217575430870056
Epoch  74 - Batch  10 - MSE: 0.16725614666938782
Epoch  75 - Batch  10 - MSE: 0.20702368021011353
Epoch  76 - Batch  10 - MSE: 0.1669454276561737
```

```
Epoch  77 - Batch  10 - MSE: 0.13878539204597473
Epoch  78 - Batch  10 - MSE: 0.20258475840091705
Epoch  79 - Batch  10 - MSE: 0.1937747746706009
Epoch  80 - Batch  10 - MSE: 0.17492397129535675
Epoch  81 - Batch  10 - MSE: 0.1957567185163498
Epoch  82 - Batch  10 - MSE: 0.17618945240974426
Epoch  83 - Batch  10 - MSE: 0.2067628800868988
Epoch  84 - Batch  10 - MSE: 0.16943976283073425
Epoch  85 - Batch  10 - MSE: 0.18528254330158234
Epoch  86 - Batch  10 - MSE: 0.18323619663715363
Epoch  87 - Batch  10 - MSE: 0.1791437268257141
Epoch  88 - Batch  10 - MSE: 0.14909659326076508
Epoch  89 - Batch  10 - MSE: 0.17638398706912994
Epoch  90 - Batch  10 - MSE: 0.1458713710308075
Epoch  91 - Batch  10 - MSE: 0.1595243215560913
Epoch  92 - Batch  10 - MSE: 0.16912803053855896
Epoch  93 - Batch  10 - MSE: 0.17969194054603577
Epoch  94 - Batch  10 - MSE: 0.19912834465503693
Epoch  95 - Batch  10 - MSE: 0.16625115275382996
Epoch  96 - Batch  10 - MSE: 0.16430987417697906
Epoch  97 - Batch  10 - MSE: 0.15476717054843903
Epoch  98 - Batch  10 - MSE: 0.19628238677978516
Epoch  99 - Batch  10 - MSE: 0.1983167976140976
Epoch 100 - Batch  10 - MSE: 0.13941439986228943
Epoch 101 - Batch  10 - MSE: 0.19001439213752747
Epoch 102 - Batch  10 - MSE: 0.17911776900291443
Epoch 103 - Batch  10 - MSE: 0.1694565862417221
Epoch 104 - Batch  10 - MSE: 0.17341981828212738
Epoch 105 - Batch  10 - MSE: 0.1694345474243164
Epoch 106 - Batch  10 - MSE: 0.13886971771717072
Epoch 107 - Batch  10 - MSE: 0.1809331476688385
Epoch 108 - Batch  10 - MSE: 0.17415301501750946
Epoch 109 - Batch  10 - MSE: 0.1793622523546219
Epoch 110 - Batch  10 - MSE: 0.20040179789066315
Epoch 111 - Batch  10 - MSE: 0.21431533992290497
Epoch 112 - Batch  10 - MSE: 0.15518257021903992
Epoch 113 - Batch  10 - MSE: 0.17341624200344086
Epoch 114 - Batch  10 - MSE: 0.17684316635131836
Epoch 115 - Batch  10 - MSE: 0.15057702362537384
Epoch 116 - Batch  10 - MSE: 0.1882852166891098
Epoch 117 - Batch  10 - MSE: 0.1639048159122467
Epoch 118 - Batch  10 - MSE: 0.17085804045200348
Epoch 119 - Batch  10 - MSE: 0.1646232157945633
Epoch 120 - Batch  10 - MSE: 0.15398059785366058
Epoch 121 - Batch  10 - MSE: 0.15777012705802917
Epoch 122 - Batch  10 - MSE: 0.16090723872184753
Epoch 123 - Batch  10 - MSE: 0.16471973061561584
Epoch 124 - Batch  10 - MSE: 0.16528114676475525
```

```
Epoch 125 - Batch  10 - MSE: 0.18746548891067505
Epoch 126 - Batch  10 - MSE: 0.18202796578407288
Epoch 127 - Batch  10 - MSE: 0.18545253574848175
Epoch 128 - Batch  10 - MSE: 0.15866854786872864
Epoch 129 - Batch  10 - MSE: 0.14892081916332245
Epoch 130 - Batch  10 - MSE: 0.18016651272773743
Epoch 131 - Batch  10 - MSE: 0.16617748141288757
Epoch 132 - Batch  10 - MSE: 0.15480898320674896
Epoch 133 - Batch  10 - MSE: 0.1607377529144287
Epoch 134 - Batch  10 - MSE: 0.18703651428222656
Epoch 135 - Batch  10 - MSE: 0.15066882967948914
Epoch 136 - Batch  10 - MSE: 0.15240977704524994
Epoch 137 - Batch  10 - MSE: 0.14806242287158966
Epoch 138 - Batch  10 - MSE: 0.17941920459270477
Epoch 139 - Batch  10 - MSE: 0.16769954562187195
Epoch 140 - Batch  10 - MSE: 0.15653932094573975
Epoch 141 - Batch  10 - MSE: 0.17573146522045135
Epoch 142 - Batch  10 - MSE: 0.1550263613462448
Epoch 143 - Batch  10 - MSE: 0.16911420226097107
Epoch 144 - Batch  10 - MSE: 0.1677565723657608
Epoch 145 - Batch  10 - MSE: 0.21703143417835236
Epoch 146 - Batch  10 - MSE: 0.15415045619010925
Epoch 147 - Batch  10 - MSE: 0.16280092298984528
Epoch 148 - Batch  10 - MSE: 0.13637511432170868
Epoch 149 - Batch  10 - MSE: 0.15500344336032867
Epoch 150 - Batch  10 - MSE: 0.14387187361717224
Epoch 151 - Batch  10 - MSE: 0.18652527034282684
Epoch 152 - Batch  10 - MSE: 0.133677676320076
Epoch 153 - Batch  10 - MSE: 0.13001002371311188
Epoch 154 - Batch  10 - MSE: 0.17910203337669373
Epoch 155 - Batch  10 - MSE: 0.16713491082191467
Epoch 156 - Batch  10 - MSE: 0.17325571179389954
Epoch 157 - Batch  10 - MSE: 0.1820521354675293
Epoch 158 - Batch  10 - MSE: 0.1671794056892395
Epoch 159 - Batch  10 - MSE: 0.20015724003314972
Epoch 160 - Batch  10 - MSE: 0.14786767959594727
Epoch 161 - Batch  10 - MSE: 0.14196893572807312
Epoch 162 - Batch  10 - MSE: 0.16898849606513977
Epoch 163 - Batch  10 - MSE: 0.17885814607143402
Epoch 164 - Batch  10 - MSE: 0.21250231564044952
Epoch 165 - Batch  10 - MSE: 0.153567373752594
Epoch 166 - Batch  10 - MSE: 0.15697018802165985
Epoch 167 - Batch  10 - MSE: 0.1730562150478363
Epoch 168 - Batch  10 - MSE: 0.19933901727199554
Epoch 169 - Batch  10 - MSE: 0.15179361402988434
Epoch 170 - Batch  10 - MSE: 0.1619376540184021
Epoch 171 - Batch  10 - MSE: 0.16080260276794434
Epoch 172 - Batch  10 - MSE: 0.1632588505744934
```

```
Epoch 173 - Batch  10 - MSE: 0.15962010622024536
Epoch 174 - Batch  10 - MSE: 0.16615618765354156
Epoch 175 - Batch  10 - MSE: 0.16713088750839233
Epoch 176 - Batch  10 - MSE: 0.19900847971439362
Epoch 177 - Batch  10 - MSE: 0.1388169378042221
Epoch 178 - Batch  10 - MSE: 0.13495014607906342
Epoch 179 - Batch  10 - MSE: 0.16761016845703125
Epoch 180 - Batch  10 - MSE: 0.15065820515155792
Epoch 181 - Batch  10 - MSE: 0.20089244842529297
Epoch 182 - Batch  10 - MSE: 0.13568554818630219
Epoch 183 - Batch  10 - MSE: 0.2252882570028305
Epoch 184 - Batch  10 - MSE: 0.13833825290203094
Epoch 185 - Batch  10 - MSE: 0.17037177085876465
Epoch 186 - Batch  10 - MSE: 0.18088003993034363
Epoch 187 - Batch  10 - MSE: 0.1801525503396988
Epoch 188 - Batch  10 - MSE: 0.18713782727718353
Epoch 189 - Batch  10 - MSE: 0.12878482043743134
Epoch 190 - Batch  10 - MSE: 0.18297971785068512
Epoch 191 - Batch  10 - MSE: 0.17273816466331482
Epoch 192 - Batch  10 - MSE: 0.14294658601284027
Epoch 193 - Batch  10 - MSE: 0.13427169620990753
Epoch 194 - Batch  10 - MSE: 0.12598194181919098
Epoch 195 - Batch  10 - MSE: 0.18607279658317566
Epoch 196 - Batch  10 - MSE: 0.13138003647327423
Epoch 197 - Batch  10 - MSE: 0.1512424498796463
Epoch 198 - Batch  10 - MSE: 0.17834371328353882
Epoch 199 - Batch  10 - MSE: 0.217040553689003
Epoch 200 - Batch  10 - MSE: 0.1460980921983719
Epoch 201 - Batch  10 - MSE: 0.2028699666261673
Epoch 202 - Batch  10 - MSE: 0.1694510281085968
Epoch 203 - Batch  10 - MSE: 0.2113415151834488
Epoch 204 - Batch  10 - MSE: 0.15999862551689148
Epoch 205 - Batch  10 - MSE: 0.1506449282169342
Epoch 206 - Batch  10 - MSE: 0.13774846494197845
Epoch 207 - Batch  10 - MSE: 0.16340374946594238
Epoch 208 - Batch  10 - MSE: 0.17314788699150085
Epoch 209 - Batch  10 - MSE: 0.19067241251468658
Epoch 210 - Batch  10 - MSE: 0.15235209465026855
Epoch 211 - Batch  10 - MSE: 0.13596954941749573
Epoch 212 - Batch  10 - MSE: 0.13022097945213318
Epoch 213 - Batch  10 - MSE: 0.168953999876976
Epoch 214 - Batch  10 - MSE: 0.14556676149368286
Epoch 215 - Batch  10 - MSE: 0.16942687332630157
Epoch 216 - Batch  10 - MSE: 0.14981865882873535
Epoch 217 - Batch  10 - MSE: 0.14419466257095337
Epoch 218 - Batch  10 - MSE: 0.16640594601631165
Epoch 219 - Batch  10 - MSE: 0.15924663841724396
Epoch 220 - Batch  10 - MSE: 0.16166435182094574
```

```
Epoch 221 - Batch  10 - MSE: 0.1643018275499344
Epoch 222 - Batch  10 - MSE: 0.18999503552913666
Epoch 223 - Batch  10 - MSE: 0.16552573442459106
Epoch 224 - Batch  10 - MSE: 0.1627187281847
Epoch 225 - Batch  10 - MSE: 0.15122252702713013
Epoch 226 - Batch  10 - MSE: 0.15112082660198212
Epoch 227 - Batch  10 - MSE: 0.18435458838939667
Epoch 228 - Batch  10 - MSE: 0.19059886038303375
Epoch 229 - Batch  10 - MSE: 0.15306374430656433
Epoch 230 - Batch  10 - MSE: 0.15099579095840454
Epoch 231 - Batch  10 - MSE: 0.18324565887451172
Epoch 232 - Batch  10 - MSE: 0.15665270388126373
Epoch 233 - Batch  10 - MSE: 0.15289010107517242
Epoch 234 - Batch  10 - MSE: 0.17920516431331635
Epoch 235 - Batch  10 - MSE: 0.17003324627876282
Epoch 236 - Batch  10 - MSE: 0.18504157662391663
Epoch 237 - Batch  10 - MSE: 0.2127557098865509
Epoch 238 - Batch  10 - MSE: 0.15622863173484802
Epoch 239 - Batch  10 - MSE: 0.1524023413658142
Epoch 240 - Batch  10 - MSE: 0.15379540622234344
Epoch 241 - Batch  10 - MSE: 0.16103637218475342
Epoch 242 - Batch  10 - MSE: 0.15858229994773865
Epoch 243 - Batch  10 - MSE: 0.13737520575523376
Epoch 244 - Batch  10 - MSE: 0.18123428523540497
Epoch 245 - Batch  10 - MSE: 0.16798816621303558
Epoch 246 - Batch  10 - MSE: 0.20221315324306488
Epoch 247 - Batch  10 - MSE: 0.15459442138671875
Epoch 248 - Batch  10 - MSE: 0.14411002397537231
Epoch 249 - Batch  10 - MSE: 0.17188438773155212
Epoch 250 - Batch  10 - MSE: 0.19021667540073395
Epoch 251 - Batch  10 - MSE: 0.1403995305299759
Epoch 252 - Batch  10 - MSE: 0.20639613270759583
Epoch 253 - Batch  10 - MSE: 0.16594621539115906
Epoch 254 - Batch  10 - MSE: 0.1805490106344223
Epoch 255 - Batch  10 - MSE: 0.1876784861087799
Epoch 256 - Batch  10 - MSE: 0.17116805911064148
Epoch 257 - Batch  10 - MSE: 0.16077306866645813
Epoch 258 - Batch  10 - MSE: 0.20374706387519836
Epoch 259 - Batch  10 - MSE: 0.17594262957572937
Epoch 260 - Batch  10 - MSE: 0.21873033046722412
Epoch 261 - Batch  10 - MSE: 0.166362926363945
Epoch 262 - Batch  10 - MSE: 0.2162914276123047
Epoch 263 - Batch  10 - MSE: 0.14846499264240265
Epoch 264 - Batch  10 - MSE: 0.1909022480249405
Epoch 265 - Batch  10 - MSE: 0.16099385917186737
Epoch 266 - Batch  10 - MSE: 0.12993726134300232
Epoch 267 - Batch  10 - MSE: 0.1613456755876541
Epoch 268 - Batch  10 - MSE: 0.15146027505397797
```

```
Epoch 269 - Batch  10 - MSE: 0.18408812582492828
Epoch 270 - Batch  10 - MSE: 0.16892626881599426
Epoch 271 - Batch  10 - MSE: 0.17531026899814606
Epoch 272 - Batch  10 - MSE: 0.16022613644599915
Epoch 273 - Batch  10 - MSE: 0.18438148498535156
Epoch 274 - Batch  10 - MSE: 0.17126698791980743
Epoch 275 - Batch  10 - MSE: 0.18412208557128906
Epoch 276 - Batch  10 - MSE: 0.13526256382465363
Epoch 277 - Batch  10 - MSE: 0.17133808135986328
Epoch 278 - Batch  10 - MSE: 0.1700066477060318
Epoch 279 - Batch  10 - MSE: 0.22967484593391418
Epoch 280 - Batch  10 - MSE: 0.15682096779346466
Epoch 281 - Batch  10 - MSE: 0.16341932117938995
Epoch 282 - Batch  10 - MSE: 0.15198618173599243
Epoch 283 - Batch  10 - MSE: 0.1858779937028885
Epoch 284 - Batch  10 - MSE: 0.16862177848815918
Epoch 285 - Batch  10 - MSE: 0.15338683128356934
Epoch 286 - Batch  10 - MSE: 0.16537483036518097
Epoch 287 - Batch  10 - MSE: 0.1974947452545166
Epoch 288 - Batch  10 - MSE: 0.12630191445350647
Epoch 289 - Batch  10 - MSE: 0.1629834920167923
Epoch 290 - Batch  10 - MSE: 0.19950221478939056
Epoch 291 - Batch  10 - MSE: 0.1946037858724594
Epoch 292 - Batch  10 - MSE: 0.1881369799375534
Epoch 293 - Batch  10 - MSE: 0.17554813623428345
Epoch 294 - Batch  10 - MSE: 0.15931373834609985
Epoch 295 - Batch  10 - MSE: 0.1587013304233551
Epoch 296 - Batch  10 - MSE: 0.16550859808921814
Epoch 297 - Batch  10 - MSE: 0.15632322430610657
Epoch 298 - Batch  10 - MSE: 0.1313316822052002
Epoch 299 - Batch  10 - MSE: 0.15559357404708862
Epoch 300 - Batch  10 - MSE: 0.16687650978565216
Epoch 301 - Batch  10 - MSE: 0.1734093576669693
Epoch 302 - Batch  10 - MSE: 0.18473312258720398
Epoch 303 - Batch  10 - MSE: 0.1618439257144928
Epoch 304 - Batch  10 - MSE: 0.1380319744348526
Epoch 305 - Batch  10 - MSE: 0.15244251489639282
Epoch 306 - Batch  10 - MSE: 0.17342250049114227
Epoch 307 - Batch  10 - MSE: 0.15935753285884857
Epoch 308 - Batch  10 - MSE: 0.15146155655384064
Epoch 309 - Batch  10 - MSE: 0.1372508853673935
Epoch 310 - Batch  10 - MSE: 0.20402874052524567
Epoch 311 - Batch  10 - MSE: 0.14825542271137238
Epoch 312 - Batch  10 - MSE: 0.1762375384569168
Epoch 313 - Batch  10 - MSE: 0.1544989049434662
Epoch 314 - Batch  10 - MSE: 0.13760623335838318
Epoch 315 - Batch  10 - MSE: 0.12614087760448456
Epoch 316 - Batch  10 - MSE: 0.15417271852493286
```

```
Epoch 317 - Batch  10 - MSE: 0.14871846139431
Epoch 318 - Batch  10 - MSE: 0.2364005446434021
Epoch 319 - Batch  10 - MSE: 0.16250339150428772
Epoch 320 - Batch  10 - MSE: 0.17926277220249176
Epoch 321 - Batch  10 - MSE: 0.17738860845565796
Epoch 322 - Batch  10 - MSE: 0.18189679086208344
Epoch 323 - Batch  10 - MSE: 0.14749981462955475
Epoch 324 - Batch  10 - MSE: 0.17428840696811676
Epoch 325 - Batch  10 - MSE: 0.2226262092590332
Epoch 326 - Batch  10 - MSE: 0.1523682326078415
Epoch 327 - Batch  10 - MSE: 0.19328582286834717
Epoch 328 - Batch  10 - MSE: 0.13918346166610718
Epoch 329 - Batch  10 - MSE: 0.1825328916311264
Epoch 330 - Batch  10 - MSE: 0.19007842242717743
Epoch 331 - Batch  10 - MSE: 0.1857742965221405
Epoch 332 - Batch  10 - MSE: 0.1521233171224594
Epoch 333 - Batch  10 - MSE: 0.1594751924276352
Epoch 334 - Batch  10 - MSE: 0.1653040498495102
Epoch 335 - Batch  10 - MSE: 0.15520402789115906
Epoch 336 - Batch  10 - MSE: 0.17119815945625305
Epoch 337 - Batch  10 - MSE: 0.19752457737922668
Epoch 338 - Batch  10 - MSE: 0.16575591266155243
Epoch 339 - Batch  10 - MSE: 0.20302550494670868
Epoch 340 - Batch  10 - MSE: 0.2177649587392807
Epoch 341 - Batch  10 - MSE: 0.1436559110879898
Epoch 342 - Batch  10 - MSE: 0.140285924077034
Epoch 343 - Batch  10 - MSE: 0.17153039574623108
Epoch 344 - Batch  10 - MSE: 0.19042198359966278
Epoch 345 - Batch  10 - MSE: 0.182703897356987
Epoch 346 - Batch  10 - MSE: 0.14573052525520325
Epoch 347 - Batch  10 - MSE: 0.19810999929904938
Epoch 348 - Batch  10 - MSE: 0.15589271485805511
Epoch 349 - Batch  10 - MSE: 0.1996491253376007
Epoch 350 - Batch  10 - MSE: 0.14397545158863068
Epoch 351 - Batch  10 - MSE: 0.20238474011421204
Epoch 352 - Batch  10 - MSE: 0.1776258498430252
Epoch 353 - Batch  10 - MSE: 0.17313043773174286
Epoch 354 - Batch  10 - MSE: 0.2001902014017105
Epoch 355 - Batch  10 - MSE: 0.1643482893705368
Epoch 356 - Batch  10 - MSE: 0.15921786427497864
Epoch 357 - Batch  10 - MSE: 0.16756701469421387
Epoch 358 - Batch  10 - MSE: 0.19492404162883759
Epoch 359 - Batch  10 - MSE: 0.19567203521728516
Epoch 360 - Batch  10 - MSE: 0.14397959411144257
Epoch 361 - Batch  10 - MSE: 0.18612071871757507
Epoch 362 - Batch  10 - MSE: 0.12549372017383575
Epoch 363 - Batch  10 - MSE: 0.17043355107307434
Epoch 364 - Batch  10 - MSE: 0.18462105095386505
```

```
Epoch 365 - Batch  10 - MSE: 0.14405560493469238
Epoch 366 - Batch  10 - MSE: 0.2005436271429062
Epoch 367 - Batch  10 - MSE: 0.20400995016098022
Epoch 368 - Batch  10 - MSE: 0.16479556262493134
Epoch 369 - Batch  10 - MSE: 0.11938998848199844
Epoch 370 - Batch  10 - MSE: 0.17983844876289368
Epoch 371 - Batch  10 - MSE: 0.19866906106472015
Epoch 372 - Batch  10 - MSE: 0.11989164352416992
Epoch 373 - Batch  10 - MSE: 0.16393031179904938
Epoch 374 - Batch  10 - MSE: 0.18946319818496704
Epoch 375 - Batch  10 - MSE: 0.17762227356433868
Epoch 376 - Batch  10 - MSE: 0.17559918761253357
Epoch 377 - Batch  10 - MSE: 0.18278899788856506
Epoch 378 - Batch  10 - MSE: 0.17486608028411865
Epoch 379 - Batch  10 - MSE: 0.15818098187446594
Epoch 380 - Batch  10 - MSE: 0.13261620700359344
Epoch 381 - Batch  10 - MSE: 0.16995295882225037
Epoch 382 - Batch  10 - MSE: 0.18351227045059204
Epoch 383 - Batch  10 - MSE: 0.17257969081401825
Epoch 384 - Batch  10 - MSE: 0.16452795267105103
Epoch 385 - Batch  10 - MSE: 0.1973210871219635
Epoch 386 - Batch  10 - MSE: 0.13632801175117493
Epoch 387 - Batch  10 - MSE: 0.1781628131866455
Epoch 388 - Batch  10 - MSE: 0.192433163523674
Epoch 389 - Batch  10 - MSE: 0.16165708005428314
Epoch 390 - Batch  10 - MSE: 0.21402208507061005
Epoch 391 - Batch  10 - MSE: 0.15525858104228973
Epoch 392 - Batch  10 - MSE: 0.1325029879808426
Epoch 393 - Batch  10 - MSE: 0.16705450415611267
Epoch 394 - Batch  10 - MSE: 0.14892394840717316
Epoch 395 - Batch  10 - MSE: 0.15407489240169525
Epoch 396 - Batch  10 - MSE: 0.15056191384792328
Epoch 397 - Batch  10 - MSE: 0.1635637879371643
Epoch 398 - Batch  10 - MSE: 0.13302522897720337
Epoch 399 - Batch  10 - MSE: 0.1494939923286438
Epoch 400 - Batch  10 - MSE: 0.1897820681333542
Epoch 401 - Batch  10 - MSE: 0.12411243468523026
Epoch 402 - Batch  10 - MSE: 0.14890716969966888
Epoch 403 - Batch  10 - MSE: 0.14228606224060059
Epoch 404 - Batch  10 - MSE: 0.16451892256736755
Epoch 405 - Batch  10 - MSE: 0.13171614706516266
Epoch 406 - Batch  10 - MSE: 0.16266539692878723
Epoch 407 - Batch  10 - MSE: 0.1650329977273941
Epoch 408 - Batch  10 - MSE: 0.1385434865951538
Epoch 409 - Batch  10 - MSE: 0.1806800812482834
Epoch 410 - Batch  10 - MSE: 0.12339024990797043
Epoch 411 - Batch  10 - MSE: 0.16778400540351868
Epoch 412 - Batch  10 - MSE: 0.16393402218818665
```

```
Epoch 413 - Batch  10 - MSE: 0.20435860753059387
Epoch 414 - Batch  10 - MSE: 0.178263857960701
Epoch 415 - Batch  10 - MSE: 0.15043427050113678
Epoch 416 - Batch  10 - MSE: 0.17763635516166687
Epoch 417 - Batch  10 - MSE: 0.16187235713005066
Epoch 418 - Batch  10 - MSE: 0.13567738234996796
Epoch 419 - Batch  10 - MSE: 0.16953876614570618
Epoch 420 - Batch  10 - MSE: 0.18533802032470703
Epoch 421 - Batch  10 - MSE: 0.13929328322410583
Epoch 422 - Batch  10 - MSE: 0.18716473877429962
Epoch 423 - Batch  10 - MSE: 0.16275134682655334
Epoch 424 - Batch  10 - MSE: 0.2104954719543457
Epoch 425 - Batch  10 - MSE: 0.21358445286750793
Epoch 426 - Batch  10 - MSE: 0.19224588572978973
Epoch 427 - Batch  10 - MSE: 0.14468270540237427
Epoch 428 - Batch  10 - MSE: 0.19169051945209503
Epoch 429 - Batch  10 - MSE: 0.1191757544875145
Epoch 430 - Batch  10 - MSE: 0.15354381501674652
Epoch 431 - Batch  10 - MSE: 0.16093900799751282
Epoch 432 - Batch  10 - MSE: 0.18092654645442963
Epoch 433 - Batch  10 - MSE: 0.1344960480928421
Epoch 434 - Batch  10 - MSE: 0.19490645825862885
Epoch 435 - Batch  10 - MSE: 0.1887393742799759
Epoch 436 - Batch  10 - MSE: 0.11276905983686447
Epoch 437 - Batch  10 - MSE: 0.15347358584403992
Epoch 438 - Batch  10 - MSE: 0.17881205677986145
Epoch 439 - Batch  10 - MSE: 0.1661592572927475
Epoch 440 - Batch  10 - MSE: 0.1699712574481964
Epoch 441 - Batch  10 - MSE: 0.18148116767406464
Epoch 442 - Batch  10 - MSE: 0.18597754836082458
Epoch 443 - Batch  10 - MSE: 0.14472268521785736
Epoch 444 - Batch  10 - MSE: 0.12602879106998444
Epoch 445 - Batch  10 - MSE: 0.15830697119235992
Epoch 446 - Batch  10 - MSE: 0.17139890789985657
Epoch 447 - Batch  10 - MSE: 0.1723662167787552
Epoch 448 - Batch  10 - MSE: 0.17972132563591003
Epoch 449 - Batch  10 - MSE: 0.141123428940773
Epoch 450 - Batch  10 - MSE: 0.1559642255306244
Epoch 451 - Batch  10 - MSE: 0.18704497814178467
Epoch 452 - Batch  10 - MSE: 0.13671107590198517
Epoch 453 - Batch  10 - MSE: 0.1495799571275711
Epoch 454 - Batch  10 - MSE: 0.18167683482170105
Epoch 455 - Batch  10 - MSE: 0.12356555461883545
Epoch 456 - Batch  10 - MSE: 0.16907858848571777
Epoch 457 - Batch  10 - MSE: 0.14194126427173615
Epoch 458 - Batch  10 - MSE: 0.1562102735042572
Epoch 459 - Batch  10 - MSE: 0.1761079877614975
Epoch 460 - Batch  10 - MSE: 0.17323213815689087
```

```
Epoch 461 - Batch  10 - MSE: 0.1777808666229248
Epoch 462 - Batch  10 - MSE: 0.22641809284687042
Epoch 463 - Batch  10 - MSE: 0.17013618350028992
Epoch 464 - Batch  10 - MSE: 0.1422959566116333
Epoch 465 - Batch  10 - MSE: 0.16917571425437927
Epoch 466 - Batch  10 - MSE: 0.12414183467626572
Epoch 467 - Batch  10 - MSE: 0.1702871322631836
Epoch 468 - Batch  10 - MSE: 0.17252278327941895
Epoch 469 - Batch  10 - MSE: 0.1521681696176529
Epoch 470 - Batch  10 - MSE: 0.17496119439601898
Epoch 471 - Batch  10 - MSE: 0.13557679951190948
Epoch 472 - Batch  10 - MSE: 0.16271933913230896
Epoch 473 - Batch  10 - MSE: 0.16245132684707642
Epoch 474 - Batch  10 - MSE: 0.17371013760566711
Epoch 475 - Batch  10 - MSE: 0.16650693118572235
Epoch 476 - Batch  10 - MSE: 0.13814926147460938
Epoch 477 - Batch  10 - MSE: 0.13412192463874817
Epoch 478 - Batch  10 - MSE: 0.1884339451789856
Epoch 479 - Batch  10 - MSE: 0.12401027232408524
Epoch 480 - Batch  10 - MSE: 0.1507880538702011
Epoch 481 - Batch  10 - MSE: 0.180315762758255
Epoch 482 - Batch  10 - MSE: 0.17684558033943176
Epoch 483 - Batch  10 - MSE: 0.16967007517814636
Epoch 484 - Batch  10 - MSE: 0.20554262399673462
Epoch 485 - Batch  10 - MSE: 0.18705759942531586
Epoch 486 - Batch  10 - MSE: 0.14628174901008606
Epoch 487 - Batch  10 - MSE: 0.20378321409225464
Epoch 488 - Batch  10 - MSE: 0.18998713791370392
Epoch 489 - Batch  10 - MSE: 0.14322997629642487
Epoch 490 - Batch  10 - MSE: 0.1907835751771927
Epoch 491 - Batch  10 - MSE: 0.1915583312511444
Epoch 492 - Batch  10 - MSE: 0.19672508537769318
Epoch 493 - Batch  10 - MSE: 0.1801026165485382
Epoch 494 - Batch  10 - MSE: 0.16596324741840363
Epoch 495 - Batch  10 - MSE: 0.13894709944725037
Epoch 496 - Batch  10 - MSE: 0.1928841769695282
Epoch 497 - Batch  10 - MSE: 0.16051974892616272
Epoch 498 - Batch  10 - MSE: 0.16116826236248016
Epoch 499 - Batch  10 - MSE: 0.1638757735490799
Epoch 500 - Batch  10 - MSE: 0.14822082221508026
```

Lets plot the losses:

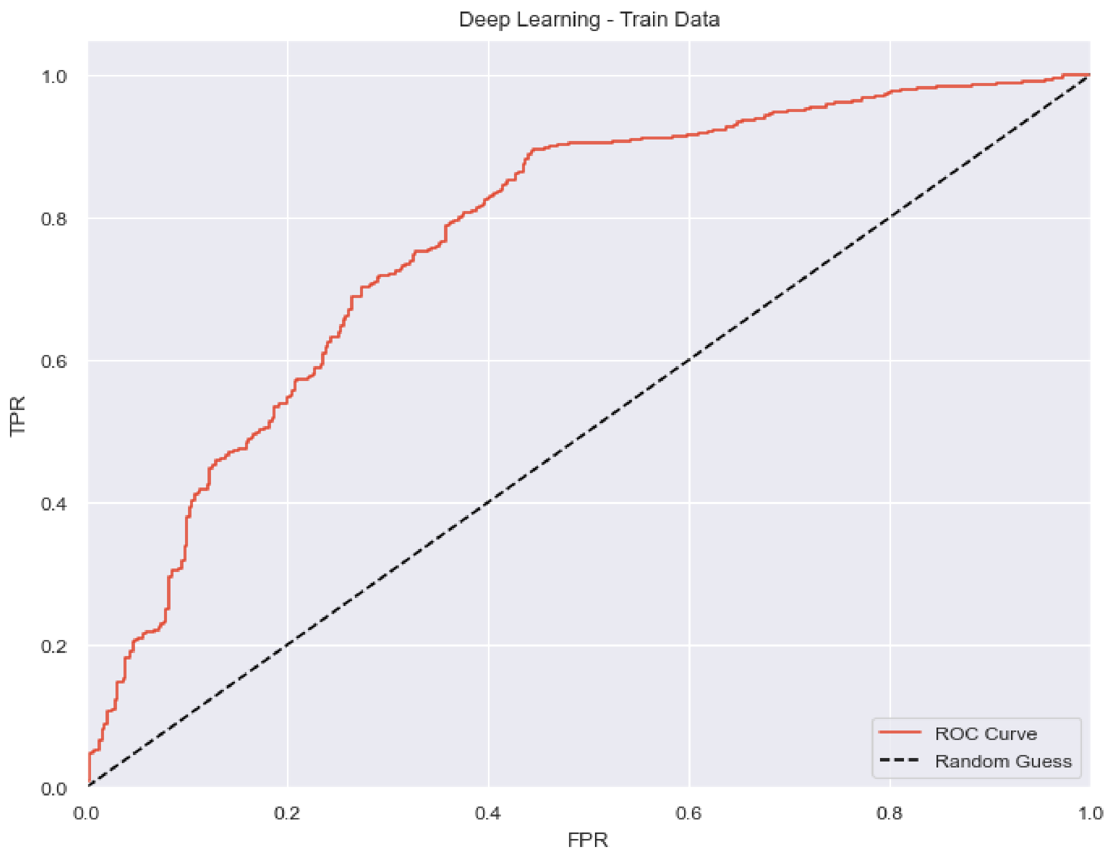
[ ]:
```
#| label: model-metrics-plots-dl-loss

plt.figure(figsize = (15, 6))
plt.plot(losses, label = "Loss")
plt.legend()
```

```
plt.show()
```

Next lets plot the ROC curve for the training data:

```
[ ]: y_train_pred_prob = model(X_train).detach().cpu().numpy()

     if torch.cuda.is_available():
         y_train = y_train.cpu()
```

```
[ ]: #| label: model-metrics-plots-dl-roc-train

     train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_train_pred_prob)

     performance_analytics_roc_curve(train_fpr, train_tpr, "Deep Learning - Train␣
       ↪Data")
```



Lets calculate the optimal threshold:

```
[ ]: optimal_threshold = round(train_thresholds[np.argmax(train_tpr - train_fpr)], 3)
     print(f'Deep Learning - Optimal Threshold: {optimal_threshold}')
```

```
Deep Learning - Optimal Threshold: 0.36000001430511475
```

With the optimal threshold we calculate the AUC for the train data:

```
[ ]: train_auc_roc = roc_auc_score(y_train, y_train_pred_prob)
     print(f'Deep Learning - AUC ROC: {train_auc_roc}')
```

```
Deep Learning - AUC ROC: 0.7720391560410812
```

Lets look at the classification report for the train data:

```
[ ]: y_train_pred_class = np.where(y_train_pred_prob <= optimal_threshold,  0, 1)
     print(classification_report(y_train, y_train_pred_class))
```

```
              precision    recall  f1-score   support

         0.0       0.71      0.56      0.62       391
         1.0       0.81      0.89      0.85       829
```

```
    accuracy                           0.79      1220
   macro avg       0.76      0.73      0.74      1220
weighted avg       0.78      0.79      0.78      1220
```

The accuracy seems good. Lets look at the confusion matrix for the train data:

```
[ ]: table = pd.crosstab(y_train_pred_class[:, 0], y_train)
     print(table)
```

```
col_0  0.0  1.0
row_0
0      218   89
1      173  740
```

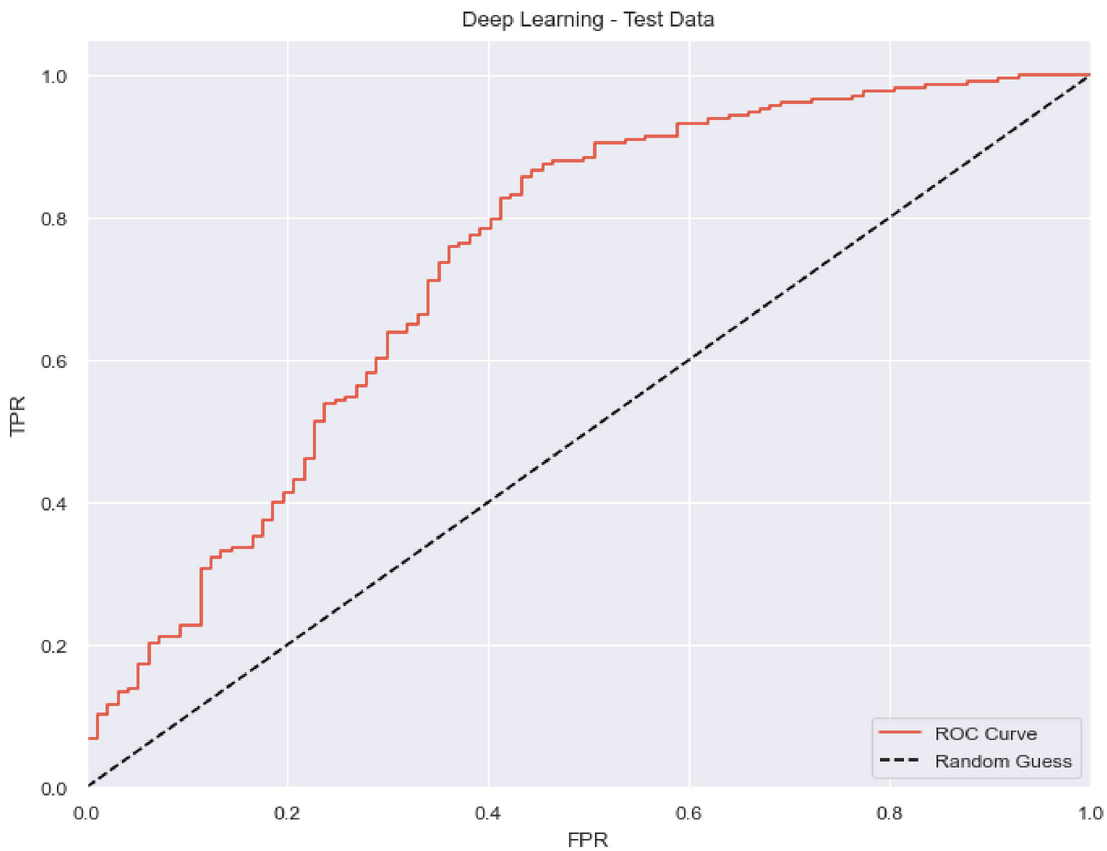
And lets look at the sensitivity and specificity for the train data:

```
[ ]: sensitivity = round((table.iloc[1, 1] / (table.iloc[0, 1] + table.iloc[1, 1]))␣
      ↪* 100, 2)
     specificity = round((table.iloc[0, 0] / (table.iloc[0, 0] + table.iloc[1, 0]))␣
      ↪* 100, 2)

     print(f"Deep Learning - Sensitivity for cut-off {optimal_threshold} is :␣
      ↪{sensitivity}%")
     print(f"Deep Learning - Specificity for cut-off {optimal_threshold} is :␣
      ↪{specificity}%")
```

```
Deep Learning - Sensitivity for cut-off 0.36000001430511475 is : 89.26%
Deep Learning - Specificity for cut-off 0.36000001430511475 is : 55.75%
```

Now lets compare with the test data - we begin with the ROC curve:

```
[ ]: #| label: model-metrics-plots-dl-roc-test

     y_test_pred_prob = model(X_test).detach().cpu().numpy()

     test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_test_pred_prob)

     performance_analytics_roc_curve(test_fpr, test_tpr, "Deep Learning - Test Data")
```



Lets calculate the AUC for the test data:

```
[ ]: test_auc_roc = roc_auc_score(y_test, y_test_pred_prob)
     print(f'Deep Learning - AUC ROC: {test_auc_roc}')
```

```
Deep Learning - AUC ROC: 0.7385507533703409
```

Lets look at the classification report for the test data:

```
[ ]: y_test_pred_class = np.where(y_test_pred_prob <= optimal_threshold,  0, 1)
     print(classification_report(y_test, y_test_pred_class))
```

```
              precision    recall  f1-score   support

         0.0       0.66      0.55      0.60        97
         1.0       0.80      0.87      0.84       208

    accuracy                           0.77       305
   macro avg       0.73      0.71      0.72       305
weighted avg       0.76      0.77      0.76       305
```

The accuracy seems good. Lets look at the confusion matrix for the test data:

```
[ ]: table = pd.crosstab(y_test_pred_class[:, 0], y_test)
     print(table)
```

```
col_0  0.0  1.0
row_0
0       53   27
1       44  181
```

And lets look at the sensitivity and specificity for the test data:

```
[ ]: sensitivity = round((table.iloc[1, 1] / (table.iloc[0, 1] + table.iloc[1, 1]))␣
      ↪* 100, 2)
     specificity = round((table.iloc[0, 0] / (table.iloc[0, 0] + table.iloc[1, 0]))␣
      ↪* 100, 2)

     print(f"Deep Learning - Sensitivity for cut-off {optimal_threshold} is :␣
      ↪{sensitivity}%")
     print(f"Deep Learning - Specificity for cut-off {optimal_threshold} is :␣
      ↪{specificity}%")
```

```
Deep Learning - Sensitivity for cut-off 0.36000001430511475 is : 87.02%
Deep Learning - Specificity for cut-off 0.36000001430511475 is : 54.64%
```

The results of the PyTorch deep learning model shows potential. The train and test AUC values are slightly inconsistent, but the accuracy values are reasonably good for little effort. It might be worth the effort to investigate this approach further.

Now we look to see if we can engineer a feature to improve the outcome.

1.5 Phase 5 - Sentiment Analysis

We now turn to Twitter / X data relating to the Nifty 50 index to see if we can mine some sentiment. First we load the tweets and create a data frame:

```
[ ]: # with open(os.path.join(os.getcwd(), "Tweets 12-6-24.txt")) as file:
     with open(os.path.join(os.getcwd(), "data/tweets/Tweets 12-6-24.txt")) as file:
         tweets = [line.rstrip() for line in file]

     data = pd.DataFrame([line for line in tweets if len(line) > 0], columns=␣
      ↪["Tweets"])
     data.head()
```

```
[ ]:                                                Tweets
     0  #bankNifty 50100 ce looks good at 70+-2 for a …
     1  "#market #banknifty #OptionsTrading #optionbuy…
     2  PENNY STOCK MADHUCON PROJECTS LTD cmp-11 FOLLO…
     3  #Nifty50 has been in a healthy uptrend since t…
     4  #Gravita #livetrading #stockstowatch #stocksin…
```

Next, we do some basic pre-processing of the data to:

1. transform all words to lowercase
2. remove all punctuation
3. remove all digits
4. remove stopwords

[ ]:
```
stop_words    = set(stopwords.words('english'))
remove_punc   = str.maketrans('', '', punctuation)
remove_digits = str.maketrans('', '', digits)

def preprocess_tweet(tweet):
    tokens = word_tokenize(tweet.lower().translate(remove_punc).
 ↪translate(remove_digits))
    return " ".join([word for word in tokens if word not in stop_words])

cleaned = data["Tweets"].apply(preprocess_tweet)
cleaned.head()
```

[ ]:
```
0           banknifty ce looks good target nifty nifty
1    market banknifty optionstrading optionbuying t…
2    penny stock madhucon projects ltd cmp followht…
3    nifty healthy uptrend since beginning year did…
4    gravita livetrading stockstowatch stocksinfocu…
Name: Tweets, dtype: object
```

Next we look at the top 20 words by frequency:

[ ]:
```
tweet_words = cleaned.str.cat(sep = " ")

freq_dist = nltk.FreqDist(tweet_words.split())
word_freq = pd.DataFrame(freq_dist.most_common(30), columns=["Word", "Freq"])
word_freq.head(30)
```

[ ]:

	Word	Freq
0	nifty	399
1	banknifty	104
2	stockmarket	71
3	niftybank	45
4	stockmarketindia	44
5	sensex	43
6	stocks	38
7	optionstrading	36
8	bse	34
9	breakoutstocks	31
10	trading	30
11	market	29
12	india	26
13	ipm	25

```
14            good    24
15             nse    24
16          growth    24
17        nseindia    23
18             may    23
19 pharmaceuticals    23
20     indianpharma    23
21      stockstobuy    22
22      sharemarket    21
23     stockmarkets    20
24              amp    19
25           points    19
26     stocksinfocus    19
27             time    19
28             bank    18
29            today    18
```

Lets add "nifty" to the list of stop words and create the cleaned tweets:

```
[ ]: stop_words = set(stopwords.words('english')) | set(["nifty"])

data["Cleaned_Tweets"] = data["Tweets"].apply(preprocess_tweet)
data.head()
```

```
[ ]:                                            Tweets  \
0  #bankNifty 50100 ce looks good at 70+-2 for a …
1  "#market #banknifty #OptionsTrading #optionbuy…
2  PENNY STOCK MADHUCON PROJECTS LTD cmp-11 FOLLO…
3  #Nifty50 has been in a healthy uptrend since t…
4  #Gravita #livetrading #stockstowatch #stocksin…

                                    Cleaned_Tweets
0                       banknifty ce looks good target
1  market banknifty optionstrading optionbuying t…
2  penny stock madhucon projects ltd cmp followht…
3  healthy uptrend since beginning year didnt bre…
4  gravita livetrading stockstowatch stocksinfocu…
```

We look again at the top 20 words by frequency:

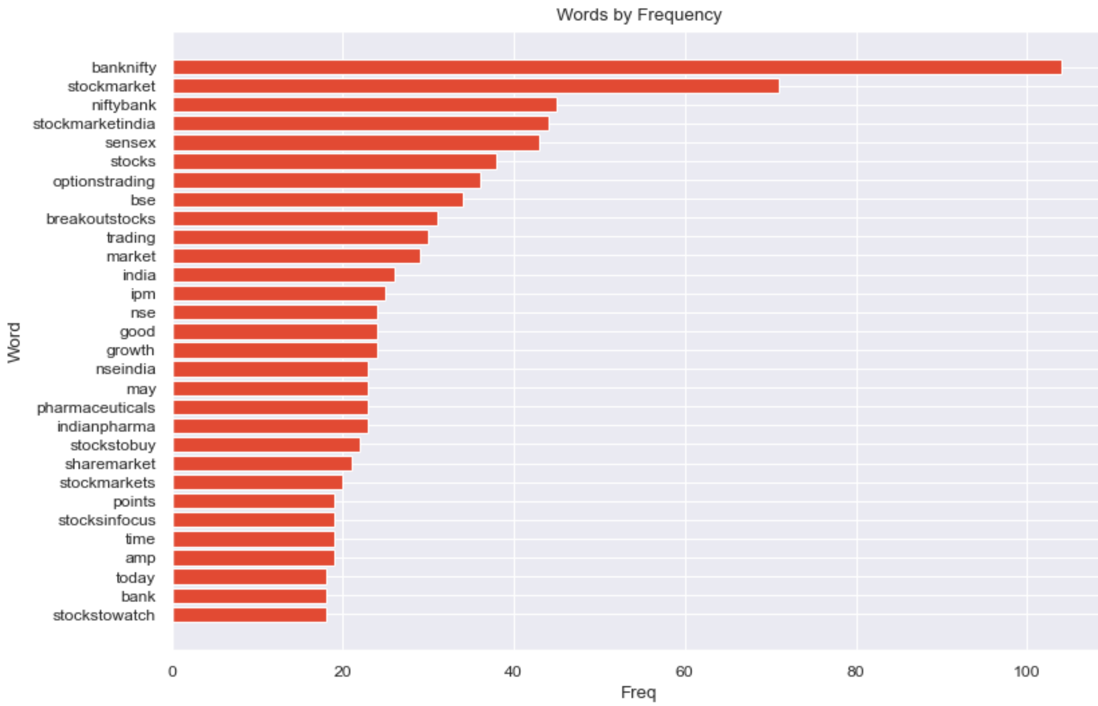
```
[ ]: tweet_words = data["Cleaned_Tweets"].str.cat(sep = " ")

freq_dist = nltk.FreqDist(tweet_words.split())
word_freq = pd.DataFrame(freq_dist.most_common(30), columns=["Word", "Freq"])
word_freq.head(30)
```

```
[ ]:              Word  Freq
0         banknifty   104
```

```
1          stockmarket    71
2            niftybank    45
3     stockmarketindia    44
4               sensex    43
5               stocks    38
6      optionstrading     36
7                  bse    34
8      breakoutstocks     31
9              trading    30
10              market    29
11               india    26
12                 ipm    25
13                good    24
14                 nse    24
15              growth    24
16           nseindia     23
17                 may    23
18     pharmaceuticals    23
19         indianpharma   23
20          stockstobuy   22
21          sharemarket   21
22         stockmarkets   20
23                 amp    19
24              points    19
25        stocksinfocus   19
26                time    19
27                bank    18
28               today    18
29        stockstowatch   18
```

We visualise these top 20 words by frequency:

[ ]:
```
#| label: tweets-words-bar-plot-freq

word_freq = word_freq.sort_values("Freq", ascending = True).reset_index(drop =␣
 ↪True)

plt.figure(figsize = (9, 6))
plt.barh(word_freq["Word"], word_freq["Freq"], align = 'center')

plt.title('Words by Frequency')
plt.xlabel('Freq')
plt.ylabel('Word')

plt.show()
```



Next, we create a word cloud:

```
[ ]: #| label: tweets-words-word-cloud

wordcloud = WordCloud(background_color = "white", collocations = False).
 ↪generate(tweet_words)

plt.figure(figsize = (9, 6))
plt.imshow(wordcloud, interpolation = "bilinear")

plt.axis("off")
plt.tight_layout(pad = 0)

plt.show()
```

Next, we extract sentiment scores for the tweets:

[ ]:
```python
sia = SentimentIntensityAnalyzer()

polarity_scores = data["Cleaned_Tweets"].apply(lambda x: sia.polarity_scores(x))

data["Positive_Score"] = polarity_scores.apply(lambda x: x["pos"])
data["Negative_Score"] = polarity_scores.apply(lambda x: x["neg"])
data["Neutral_Score"]  = polarity_scores.apply(lambda x: x["neu"])
data["Compound_Score"] = polarity_scores.apply(lambda x: x["compound"])

data.head()
```

[ ]:
```
                                              Tweets  \
0  #bankNifty 50100 ce looks good at 70+-2 for a …
1  "#market #banknifty #OptionsTrading #optionbuy…
2  PENNY STOCK MADHUCON PROJECTS LTD cmp-11 FOLLO…
3  #Nifty50 has been in a healthy uptrend since t…
4  #Gravita #livetrading #stockstowatch #stocksin…

                                      Cleaned_Tweets  Positive_Score  \
0                       banknifty ce looks good target           0.420
1  market banknifty optionstrading optionbuying t…          0.075
2  penny stock madhucon projects ltd cmp followht…          0.155
3  healthy uptrend since beginning year didnt bre…          0.100
4  gravita livetrading stockstowatch stocksinfocu…          0.262

   Negative_Score  Neutral_Score  Compound_Score
```

```
0           0.000           0.580           0.4404
1           0.145           0.780          -0.3400
2           0.000           0.845           0.2960
3           0.198           0.702          -0.3935
4           0.000           0.738           0.5994
```

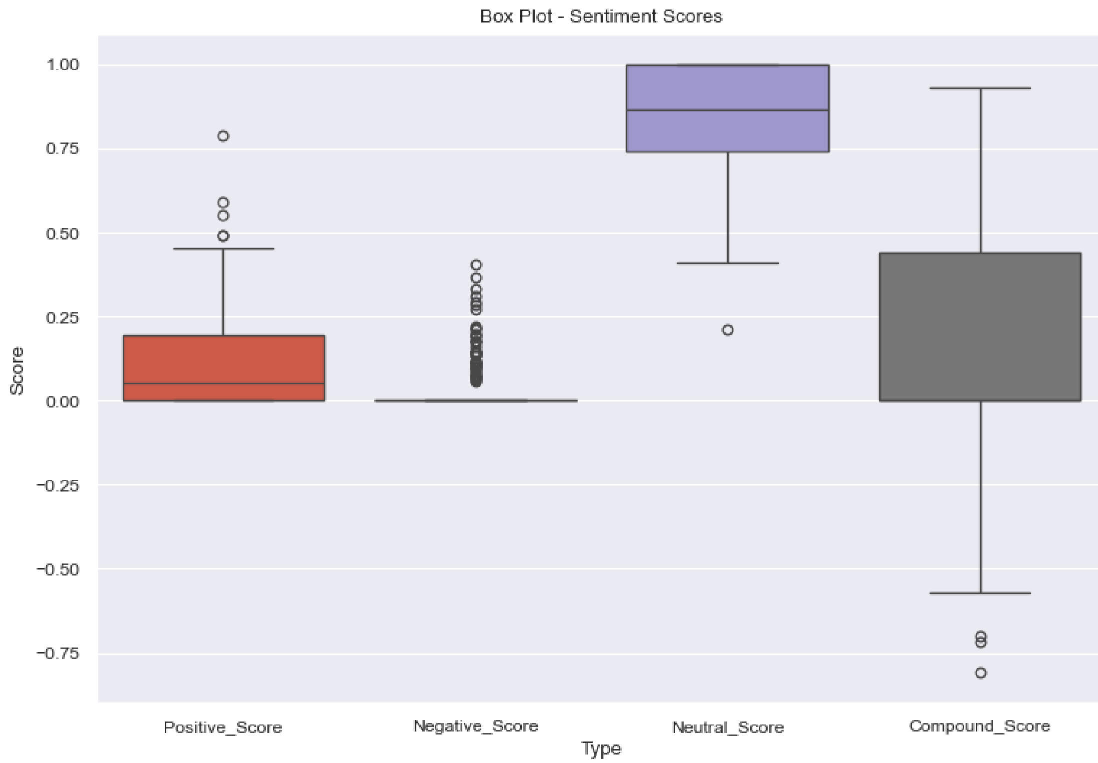
We look at the summary statistics for the sentiment scores:

```
[ ]: scores = data[["Positive_Score", "Negative_Score", "Neutral_Score",␣
      ↪"Compound_Score"]]
     scores.describe()
```

```
[ ]:        Positive_Score  Negative_Score  Neutral_Score  Compound_Score
     count      245.000000      245.000000     245.000000      245.000000
     mean         0.116216        0.028490       0.855314        0.172913
     std          0.146029        0.071052       0.156892        0.343955
     min          0.000000        0.000000       0.213000       -0.807400
     25%          0.000000        0.000000       0.742000        0.000000
     50%          0.053000        0.000000       0.868000        0.000000
     75%          0.194000        0.000000       1.000000        0.440400
     max          0.787000        0.405000       1.000000        0.928700
```

The compound score has a reasonable spread with a median of 0. We plot the sentiment scores below:

```
[ ]: #| label: tweets-words-sentiment-scores-box-plot

     plt.figure(figsize = (9, 6))
     sns.boxplot(data = scores)

     plt.title("Box Plot - Sentiment Scores")
     plt.xlabel("Type")
     plt.ylabel("Score")

     plt.show()
```



It looks like the sentiment scores could be useful as an extra feature - but without access to historical tweets, it would be impossible to tell conclusively, but it could be worth investigating further.

1.6 Conclusion

predicting NSEI open direction looks very promissing. We have shown the ability to train models with accuracy of around 75% in the case of SVM, and 76% in the case of PyTorch deep learning model - I am confident that with enhanced model tuning, and by adding extra features such as sentiment, we can improve accuracy significantly.