In [55]:
```python
# Import packages

# Data manipulation
import pandas as pd
import numpy as np
import pandas_datareader as data

# Plotting
import matplotlib
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

# Finance related operations
from pandas_datareader import data as pdf
import yfinance as yfin

# Import this to silence a warning when converting data column of a datafram
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()

# Sklearn

from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV

from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV

from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures

from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier

from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from tabulate import tabulate
```

AC 209A Final Project

Predicting Stock Variation Using Financial Indicators

Sabrina Hu and Hamzeh Hamdan
Harvard University
Fall 2022

Table of Contents

Introduction and Main Objectives

Return to contents

Predicting the movements of stocks over time based on various financial and economic indicators is a common task in the field of finance and investment --- investors and traders rely on these predictions to assess when to buy, sell, and hold stocks, and also to gain a competitive edge in the markets and maximize returns while managing risk. However, predicting the movement of stocks is not an easy task, as prices are constantly fluctuating based on many complex factors like economic indicators, corporate earnings reports, geopolitical events, and market sentiment. In this project, our objective is to understand the relationship between various financial indicators and the increase or decrease in value of a stock.

The key research questions guiding our project include:

- Which financial indicators are the best predictors of stock price/stock price increase or decrease?
- What type of model can best use financial indicators to classify the increase or decrease of a stock?

The data we will use is the 2018 data from this Kaggle dataset: ‘200+ Financial Indicators of U.S. stocks (2014-2018)’: https://www.kaggle.com/datasets/cnic92/200-financial-indicators-of-us-stocks-20142018?select=2018_Financial_Data.csv

We will first conduct some exploratory data analysis of this dataset before proceeding with the models.

Summary of the Data

Return to contents

First, we load the data as a pandas dataframe and drop rows with no information. We find that the data has 4392 samples and 224 columns. Of these, 222 are numeric, 1 is an int type, and 1 is an object type.

These are:

- 222 numeric: financial indicators
- 1 integer: class column (1 = positive stock price variation, 0 = negative stock price variation)
- 1 object: categorical name of the sector

In [56]:
```
# Create a pandas DF of the data, making sure that the stock ticker is the i
df = pd.read_csv('data/2018_Financial_Data.csv', index_col=0)

# Drop rows with no information
df.dropna(how='all', inplace=True)

df = df.loc[:, ~df.columns.str.contains('^Unnamed')]

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4392 entries, CMCSA to ZYME
Columns: 224 entries, Revenue to Class
dtypes: float64(222), int64(1), object(1)
memory usage: 7.5+ MB
```

In [57]:
```
# Shape of dataset
print(df.shape)
```

```
(4392, 224)
```

In [58]:
```
print("Features of the dataset and their data types:")
print(df.dtypes)
```

```
Features of the dataset and their data types:
Revenue                  float64
Revenue Growth           float64
Cost of Revenue          float64
Gross Profit             float64
R&D Expenses             float64
                          ...
R&D Expense Growth       float64
SG&A Expenses Growth     float64
Sector                    object
2019 PRICE VAR [%]       float64
Class                      int64
Length: 224, dtype: object
```

In [59]:
```
# Describe dataset variables
df.describe()
```

Out[59]:

	Revenue	Revenue Growth	Cost of Revenue	Gross Profit	R& Expens
count	4.346000e+03	4253.000000	4.207000e+03	4.328000e+03	4.155000e+
mean	5.119287e+09	3.455278	3.144946e+09	2.043954e+09	1.180176e+
std	2.049504e+10	195.504906	1.508813e+10	7.682369e+09	9.330891e+
min	-6.894100e+07	-3.461500	-2.669055e+09	-1.818220e+09	-1.042000e+
25%	6.501425e+07	0.000000	3.415500e+06	3.618903e+07	0.000000e+
50%	4.982640e+08	0.074900	1.741180e+08	2.219470e+08	0.000000e+
75%	2.457878e+09	0.188500	1.297814e+09	9.767015e+08	1.450150e+
max	5.003430e+11	12739.000000	3.733960e+11	1.269470e+11	2.883700e+

8 rows × 223 columns

Exploratory Data Analysis and Data Cleaning

Return to contents

Exploratory Data Analysis

First, we will do some exploratory data analysis of our dataset to extract important insights and learn how the variables are related to each other.

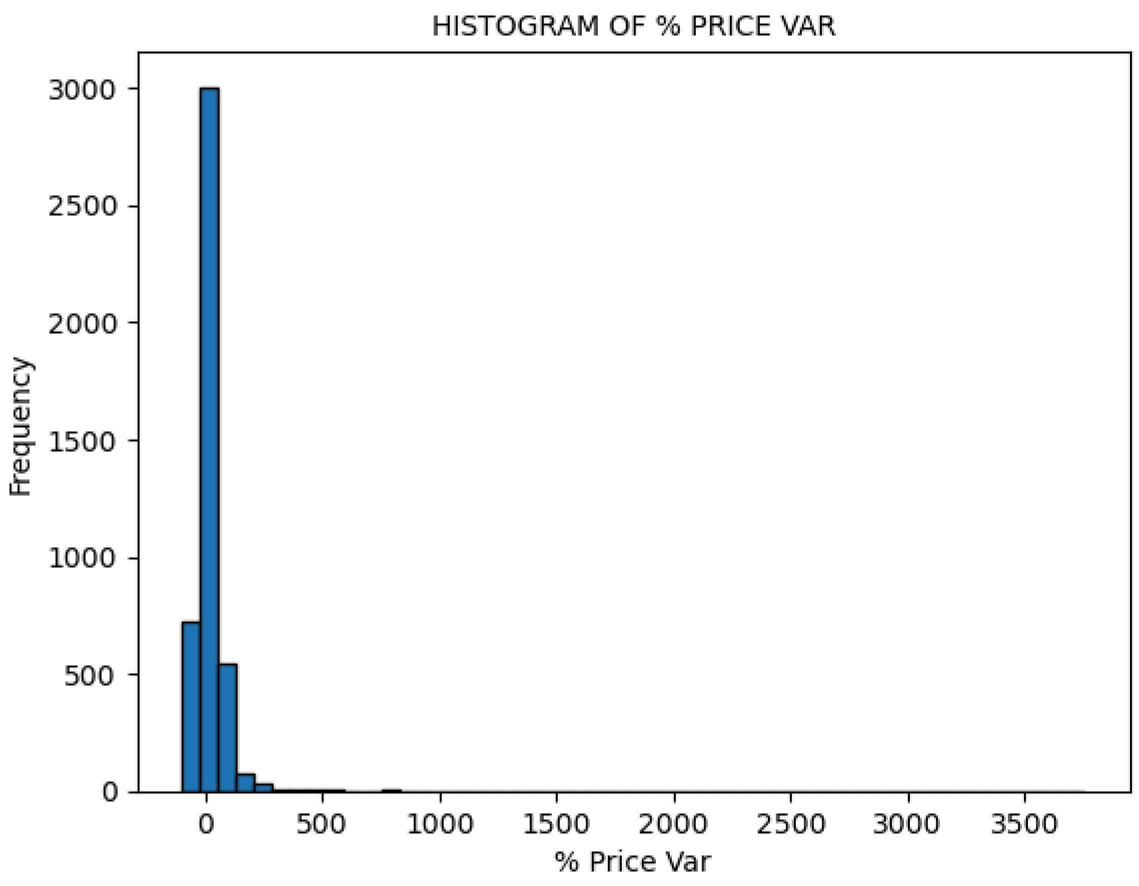
We plotted the % price variation for all observations, and also specifically by sector. We also plotted bar graphs of class counts (1 = positive price variation, 0 = negative price variation) and sector counts. Here are some key observations:

- The % Price Var mainly centers around 0, which shows that most stocks do not significantly increase or decrease. This may tell us that the classification problem in this case may be a better problem to pursue than predicting the actual value of the stock variation.
- When plotting the % Price Var by sector, we see that all sectors generally center around 0, which follows the overall trend. However, real estate and utilities are the only sectors where the $ price variation doesn't center around 0, which may indicate that being in those sectors is correlated with positive % price variation.
- The samples are not balanced in terms of class. This is very important and should be accounted for when splitting the data between the training and testing data. We have approximately 3000 stocks of class count 1 (they are buy-worthy stocks that had a positive annual return), and just under 1500 of class count 0 (they are not buy-worthy stocks as they had a negative. annual return).
- The sectors are not equally represented in the data. There are 5 sectors with 500+ stocks and the remanining 6 have less than 300 stocks. Of these, 2 have less than 100 stocks. This should be kept in mind as we choose our model.

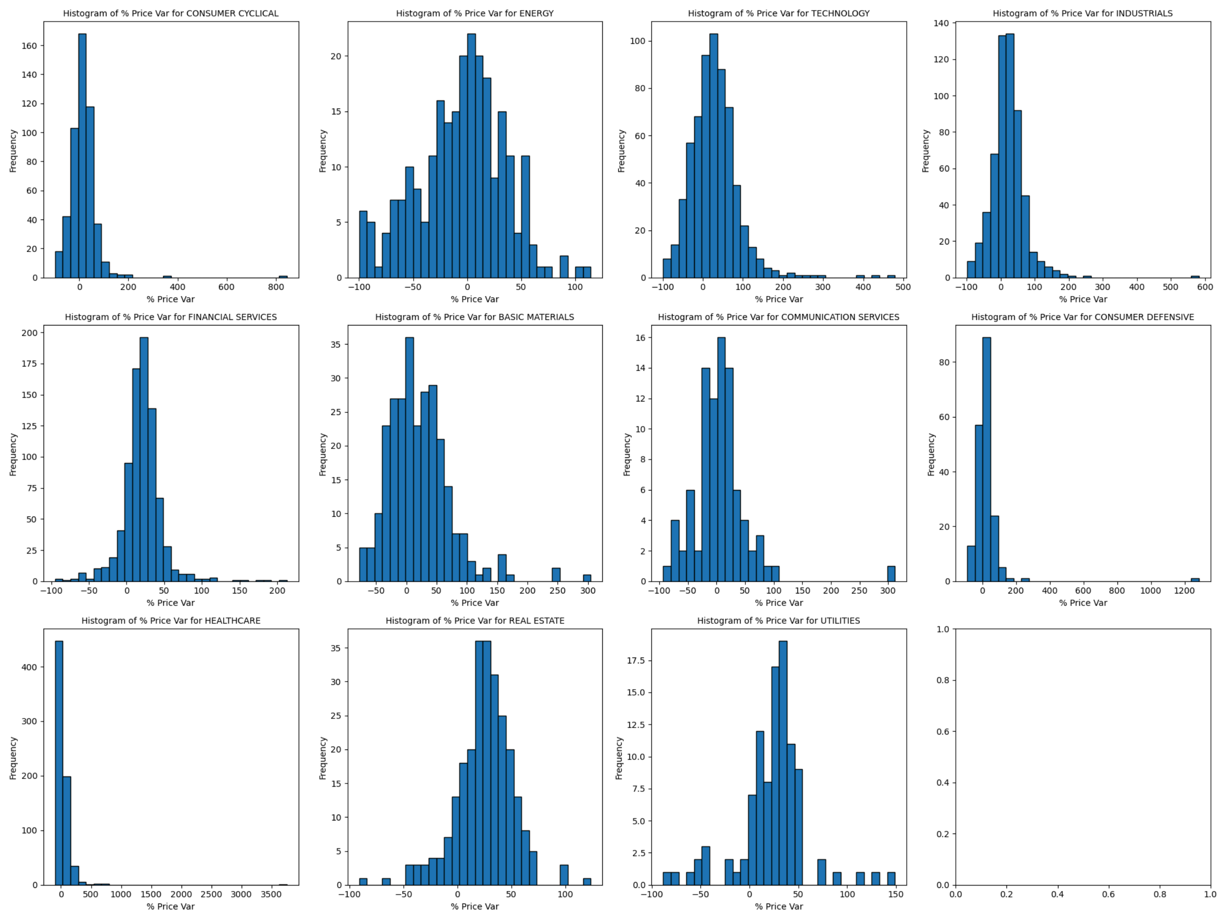
We then try to find errors in the data by plotting the annual price variation of each stock in a sector along a graph. Very high values can be errors in data entry or simply inorganic growth in returns. We found that these sectors had at least one stock with over ~500% returns, and likely are not realistic: Consumer Cyclical, Technology, Industrials, Consumer Defensive, and Healthcare. Note that this choice was arbitrary.

There were a total of 8 stocks under this category. For each one, we plotted the daily close value for the year along with the volume. This helps us find out if the gains are organic or are due to an error. We find that of those, DRIO seems to not be organic as the price jumps x8 very quickly, then doubles yet again within a month. We dropped this stock from the data. We also drop the 4 stocks that have been delisted from the data.

In [60]:
```
# Plot price variation distribution
plt.hist(df['2019 PRICE VAR [%]'], bins=50, edgecolor='black')
plt.title('HISTOGRAM OF % PRICE VAR', fontsize=10)
plt.xlabel('% Price Var')
plt.ylabel('Frequency')
plt.show()
```

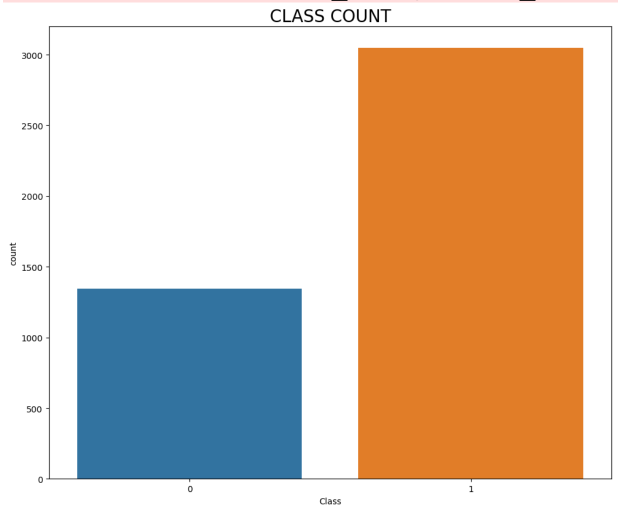


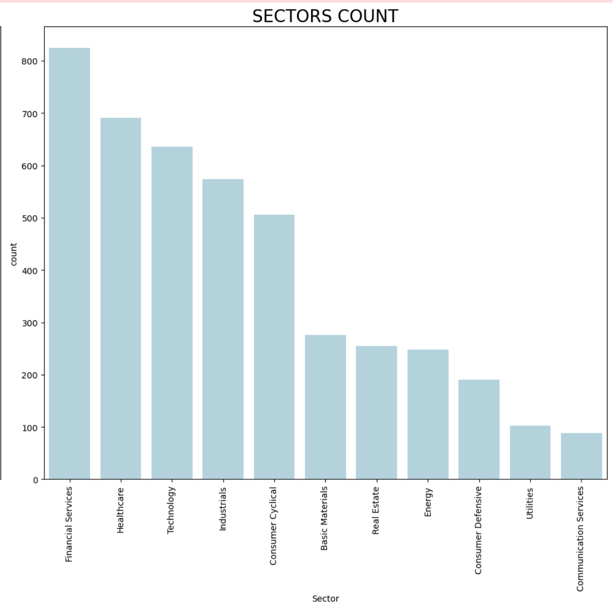
In [61]:
```
# Extract the columns we need in this step from the dataframe
df_ = df.loc[:, ['Sector', '2019 PRICE VAR [%]']]

# Get list of sectors
sector_list = df_['Sector'].unique()

# Create a 3x4 grid for the histograms
fig, axs = plt.subplots(3, 4, figsize=(20, 15))
axs = axs.ravel()

# Plot the histogram of percent price variation for each sector
for i, sector in enumerate(sector_list):

    temp = df_[df_['Sector'] == sector]

    axs[i].hist(temp['2019 PRICE VAR [%]'], bins=30, edgecolor='black')
    axs[i].set_title('Histogram of % Price Var for ' + sector.upper(), fonts
    axs[i].set_xlabel('% Price Var')
    axs[i].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```



In [62]:
```
# Create a 1x2 grid for the subplots
fig, axs = plt.subplots(1, 2, figsize=(20, 10))

# Plot class distribution
df_class = df['Class'].value_counts()
sns.barplot(x=df_class.index, y=df_class, ax=axs[0])
axs[0].set_title('CLASS COUNT', fontsize=20)

# Plot sector distribution
df_sector = df['Sector'].value_counts()
sns.barplot(x=df_sector.index, y=df_sector, color='lightblue', ax=axs[1])
axs[1].set_xticks(np.arange(len(df_sector)))
axs[1].set_xticklabels(df_sector.index.values.tolist(), rotation=90)
axs[1].set_title('SECTORS COUNT', fontsize=20)

plt.tight_layout()
plt.show()
```

```
/Users/Sabrina/micromamba/envs/cs109a/lib/python3.11/site-packages/seaborn/_
oldcore.py:1498: FutureWarning: is_categorical_dtype is deprecated and will
be removed in a future version. Use isinstance(dtype, CategoricalDtype) inst
ead
  if pd.api.types.is_categorical_dtype(vector):
/Users/Sabrina/micromamba/envs/cs109a/lib/python3.11/site-packages/seaborn/_
oldcore.py:1498: FutureWarning: is_categorical_dtype is deprecated and will
be removed in a future version. Use isinstance(dtype, CategoricalDtype) inst
ead
  if pd.api.types.is_categorical_dtype(vector):
/Users/Sabrina/micromamba/envs/cs109a/lib/python3.11/site-packages/seaborn/_
oldcore.py:1498: FutureWarning: is_categorical_dtype is deprecated and will
be removed in a future version. Use isinstance(dtype, CategoricalDtype) inst
ead
  if pd.api.types.is_categorical_dtype(vector):
/Users/Sabrina/micromamba/envs/cs109a/lib/python3.11/site-packages/seaborn/_
oldcore.py:1498: FutureWarning: is_categorical_dtype is deprecated and will
be removed in a future version. Use isinstance(dtype, CategoricalDtype) inst
ead
  if pd.api.types.is_categorical_dtype(vector):
/Users/Sabrina/micromamba/envs/cs109a/lib/python3.11/site-packages/seaborn/_
oldcore.py:1498: FutureWarning: is_categorical_dtype is deprecated and will
be removed in a future version. Use isinstance(dtype, CategoricalDtype) inst
ead
  if pd.api.types.is_categorical_dtype(vector):
/Users/Sabrina/micromamba/envs/cs109a/lib/python3.11/site-packages/seaborn/_
oldcore.py:1498: FutureWarning: is_categorical_dtype is deprecated and will
be removed in a future version. Use isinstance(dtype, CategoricalDtype) inst
ead
  if pd.api.types.is_categorical_dtype(vector):
```

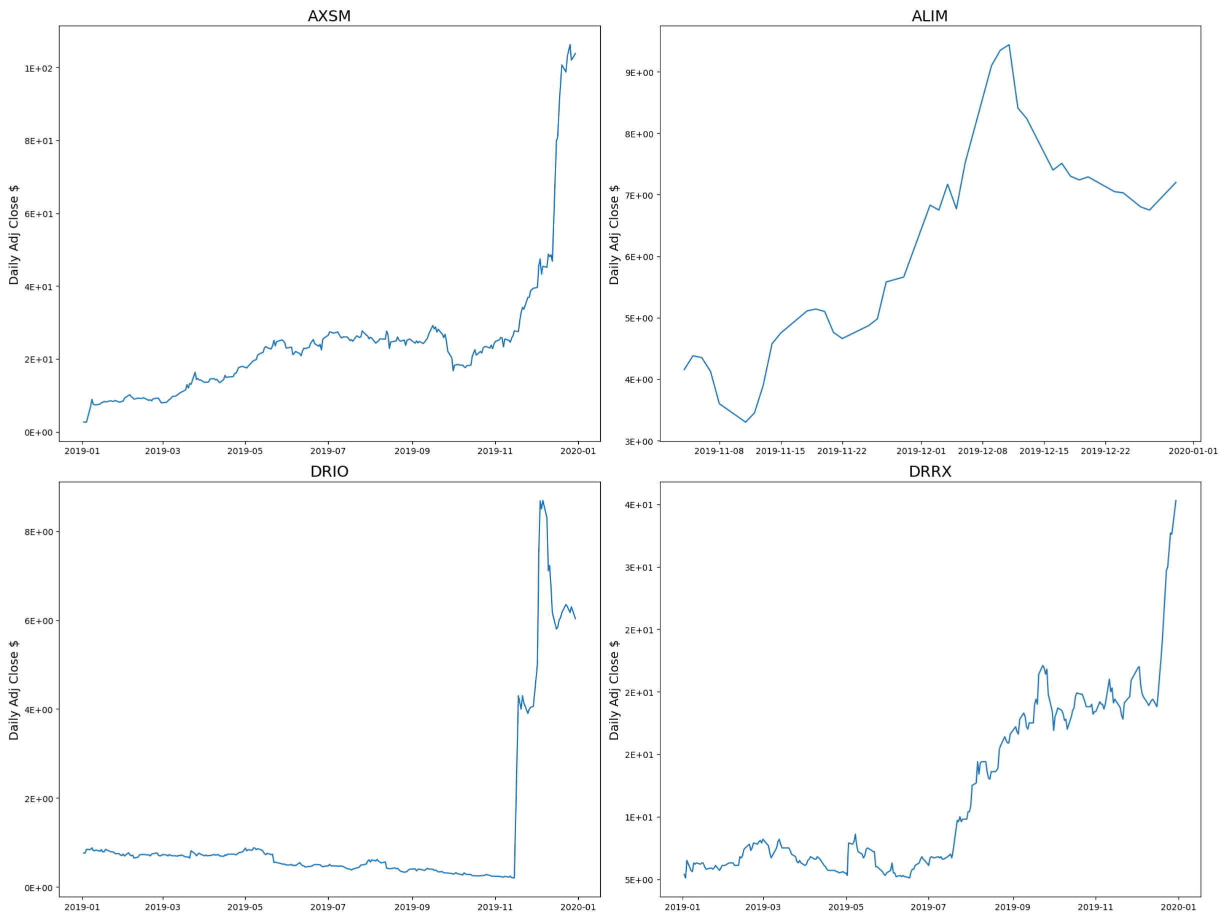




In [63]:
```python
# Get stocks that increased more than 500%
gain = 500
top_gainers = df_[df_['2019 PRICE VAR [%]'] >= gain]
top_gainers = top_gainers['2019 PRICE VAR [%]'].sort_values(ascending=False)
print(f'{len(top_gainers)} STOCKS with more than {gain}% gain.')

# Set
```

```
date_start = '2019-01-01'
date_end = '2019-12-31'
tickers = top_gainers.index.values.tolist()

# Create a 2x2 grid for the subplots
fig, axs = plt.subplots(2, 2, figsize=(20, 15))
axs = axs.ravel()

j = 0
for i, ticker in enumerate(tickers):
    try:

        yfin.pdr_override()
        # Pull daily prices for each ticker from Yahoo Finance
        daily_price = pdf.get_data_yahoo(ticker, start=date_start, end=date_

        # Check if data download was successful
        if not daily_price.empty:
            # Plot prices with volume
            axs[j].plot(daily_price['Adj Close'])
            axs[j].set_title(ticker, fontsize=18)
            axs[j].set_ylabel('Daily Adj Close $', fontsize=14)
            axs[j].yaxis.set_major_formatter(matplotlib.ticker.StrMethodForm
            j += 1

        else:
            print(f"Data not available for {ticker}. Skipping plot.")

    except Exception as e:
        print(f"Error fetching data for {ticker}: {e}")

fig.tight_layout()
plt.show()
```

```
8 STOCKS with more than 500% gain.
[*********************100%%**********************]  1 of 1 completed
[*********************100%%**********************]  1 of 1 completed

1 Failed download:
['ANFI']: Exception('%ticker%: No timezone found, symbol may be delisted')
Data not available for ANFI. Skipping plot.
[*********************100%%**********************]  1 of 1 completed

1 Failed download:
['SSI']: Exception('%ticker%: No timezone found, symbol may be delisted')
Data not available for SSI. Skipping plot.
[*********************100%%**********************]  1 of 1 completed
[*********************100%%**********************]  1 of 1 completed
[*********************100%%**********************]  1 of 1 completed
[*********************100%%**********************]  1 of 1 completed

1 Failed download:
['ARQL']: Exception('%ticker%: No timezone found, symbol may be delisted')
Data not available for ARQL. Skipping plot.
[*********************100%%**********************]  1 of 1 completed

1 Failed download:
['HEBT']: Exception('%ticker%: No timezone found, symbol may be delisted')
Data not available for HEBT. Skipping plot.
```



```
In [64]: delisted_stocks = ['ANFI', 'SSI', 'ARQL', 'HEBT']
         df_ = df_.drop(delisted_stocks + ['DRIO'], errors='ignore')
```

Data Cleaning - Missing Values

After doing a quick check for missing values, we found that there were many variables with a lot of missing values or 0 values. We plotted this as a percentage and filtered through predictors using this method.

We used an ~6% nan and zeros dominance threshold to drop columns (so that each feature is at most ~6% nan values and at most ~6% zero values). This resulted in dropping the top 50% nan-dominant financial indicators, and the top 40% zero-dominant financial indicators. This resulted in 70 remaining predictors of the 224.

Then, we deleted the top 3 and bottom 3 percentiles of the data in order to account for outliers.

Finally, we imputed missing values with the mean of the column's sector, given the underlying characteristics of the stocks in each sector. This should not have a large impact on our analysis since this was only ~6% of all data in the columns.

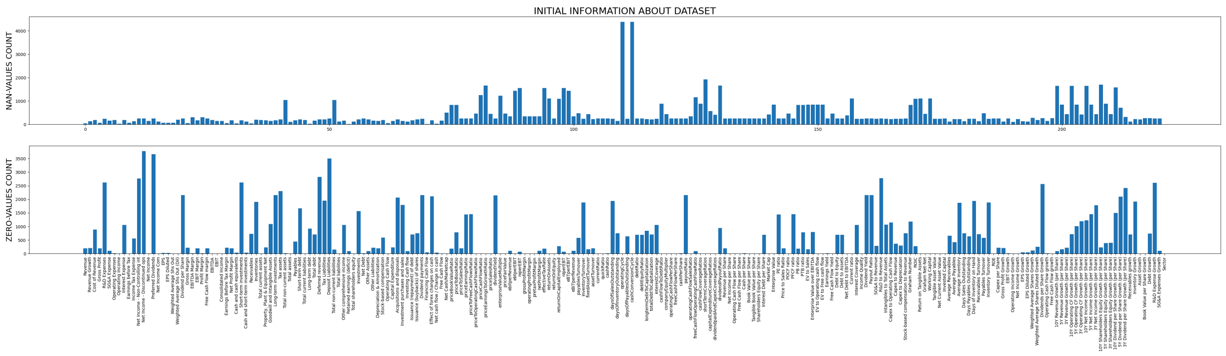
```
In [65]: missing_values = df.isnull().sum()

         missing_df = pd.DataFrame({'Column': missing_values.index, 'Missing Values':

         missing_df = missing_df.sort_values(by='Missing Values', ascending=False)

         print(missing_df)
```

```
                                          Column  Missing Values
112                        cashConversionCycle            4386
110                             operatingCycle            4386
127                    shortTermCoverageRatios            1926
208  10Y Shareholders Equity Growth (per Share)            1695
82                  priceEarningsToGrowthRatio            1658
..                                         ...             ...
54                  Retained earnings (deficit)              21
70                         Financing Cash Flow              19
221                                     Sector               0
222                           2019 PRICE VAR [%]               0
223                                      Class               0

[224 rows x 2 columns]
```

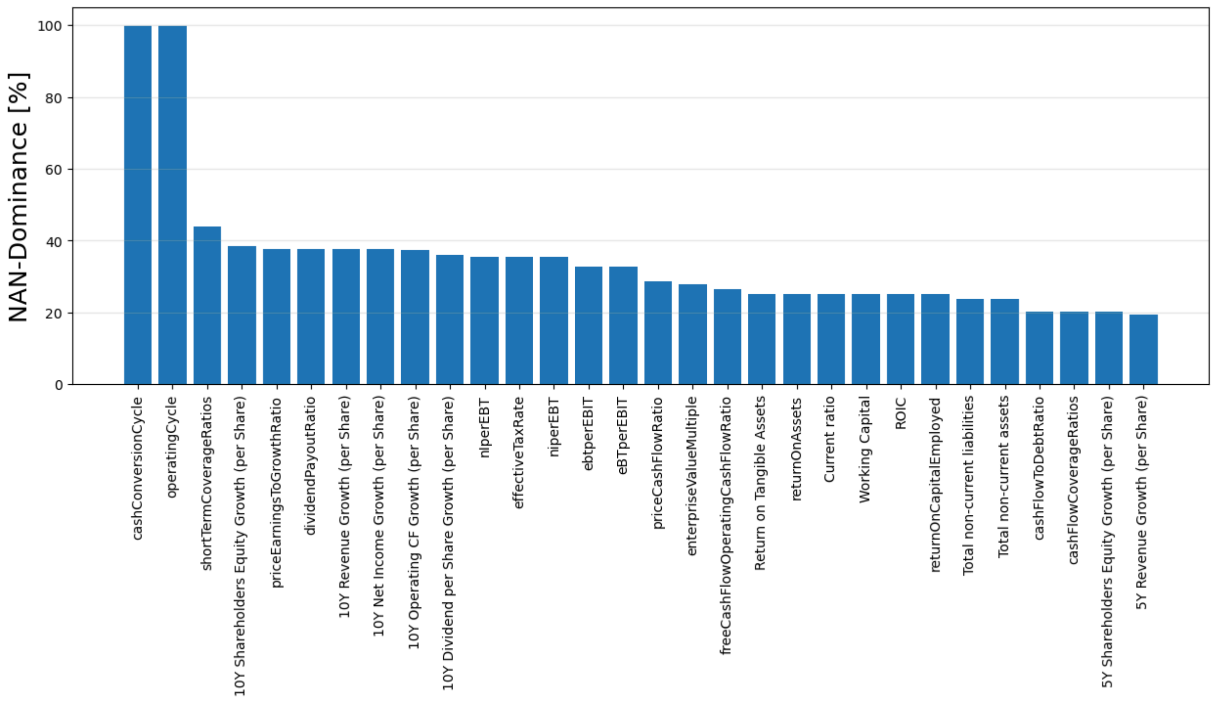
From the above, we observe that the two columns with the most missingness are 'cashConversionCycle' and 'operatingCycle', both other which are missing almost all of their values. Although there are many reasons for why certain indicators might be missing so much data, whether it be through data collection methods or the actual companies' records, in this case, it is likely that the indicators with so much missingness are not collected as rigorously for a reason. Thus, it is much more practical to remove these indicators, especially since we have 222 to begin with.

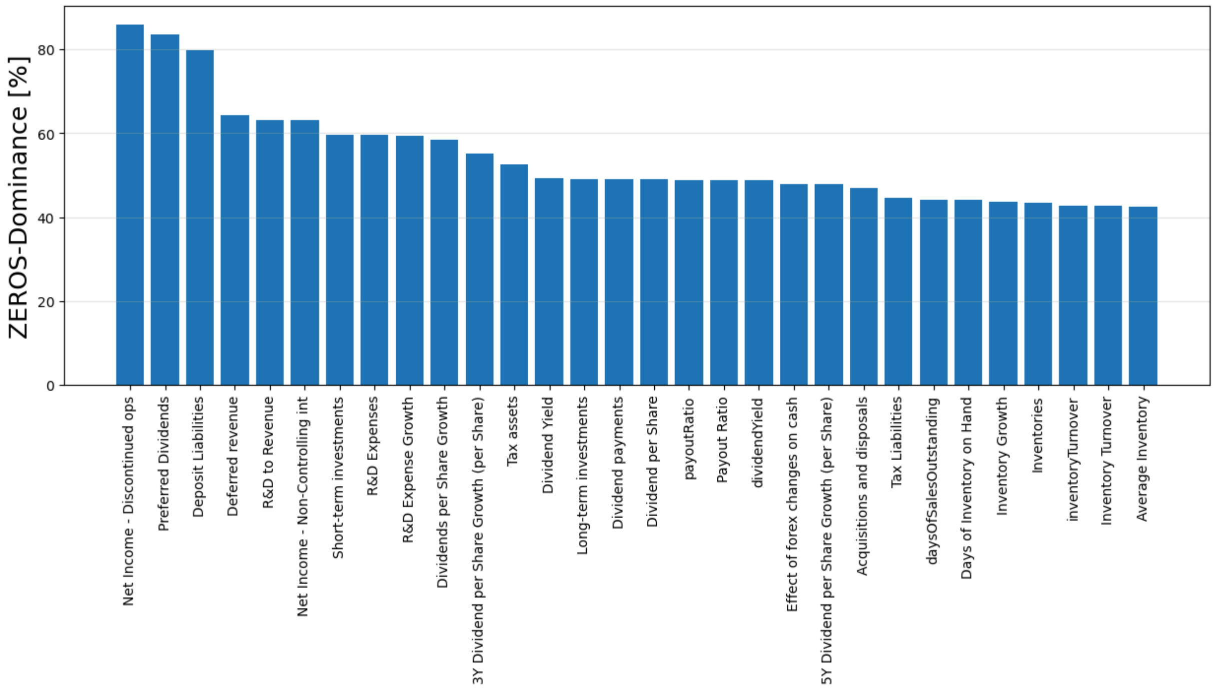
Below, we plotted both the NAN-values and zero-values count for all of the indicators, just to given a picture of how much missingness there is overall in the dataset. We also plotted the indicators with the highest NAN and zero dominance.

```
In [66]: # Drop columns relative to classification, we will use them later
         class_data = df.loc[:, ['Class', '2019 PRICE VAR [%]']]
         df.drop(['Class', '2019 PRICE VAR [%]'], inplace=True, axis=1)

         # Plot initial status of data quality in terms of nan-values and zero-values
         nan_vals = df.isna().sum()
         zero_vals = df.isin([0]).sum()
         ind = np.arange(df.shape[1])

         plt.figure(figsize=(50,10))

         plt.subplot(2,1,1)
         plt.title('INITIAL INFORMATION ABOUT DATASET', fontsize=22)
         plt.bar(ind, nan_vals.values.tolist())
         plt.ylabel('NAN-VALUES COUNT', fontsize=18)
         plt.subplot(2,1,2)
         plt.bar(ind, zero_vals.values.tolist())
         plt.ylabel('ZERO-VALUES COUNT', fontsize=18)
         plt.xticks(ind, nan_vals.index.values, rotation=90)

         plt.show()
```



In [67]:
```
# Find count and percent of nan-values, zero-values
total_nans = df.isnull().sum().sort_values(ascending=False)
percent_nans = (df.isnull().sum()/df.isnull().count() * 100).sort_values(asc
total_zeros = df.isin([0]).sum().sort_values(ascending=False)
percent_zeros = (df.isin([0]).sum()/df.isin([0]).count() * 100).sort_values(
df_nans = pd.concat([total_nans, percent_nans], axis=1, keys=['Total NaN', '
df_zeros = pd.concat([total_zeros, percent_zeros], axis=1, keys=['Total Zero

# Graphical representation
plt.figure(figsize=(15,5))
plt.bar(np.arange(30), df_nans['Percent NaN'].iloc[:30].values.tolist())
plt.xticks(np.arange(30), df_nans['Percent NaN'].iloc[:30].index.values.toli
plt.ylabel('NAN-Dominance [%]', fontsize=18)
plt.grid(alpha=0.3, axis='y')
plt.show()

plt.figure(figsize=(15,5))
plt.bar(np.arange(30), df_zeros['Percent Zeros'].iloc[:30].values.tolist())
plt.xticks(np.arange(30), df_zeros['Percent Zeros'].iloc[:30].index.values.t
plt.ylabel('ZEROS-Dominance [%]', fontsize=18)
plt.grid(alpha=0.3, axis='y')
plt.show()
```





In [68]:
```
# Find reasonable threshold for nan-values situation
test_nan_level = 0.5
print(df_nans.quantile(test_nan_level))
_, thresh_nan = df_nans.quantile(test_nan_level)

# Find reasonable threshold for zero-values situation
test_zeros_level = 0.6
print(df_zeros.quantile(test_zeros_level))
_, thresh_zeros = df_zeros.quantile(test_zeros_level)
```

```
Total NaN      251.000000
Percent NaN      5.714936
Name: 0.5, dtype: float64
Total Zeros      282.600000
Percent Zeros      6.434426
Name: 0.6, dtype: float64
```

In [69]:
```
# Clean dataset applying thresholds for both zero values, nan-values
print(f'INITIAL NUMBER OF VARIABLES: {df.shape[1]}')
print()

df_test1 = df.drop((df_nans[df_nans['Percent NaN'] > thresh_nan]).index, axi
print(f'NUMBER OF VARIABLES AFTER NaN THRESHOLD 6%: {df_test1.shape[1]}')
print()

df_zeros_postnan = df_zeros.drop((df_nans[df_nans['Percent NaN'] > thresh_na
df_test2 = df_test1.drop((df_zeros_postnan[df_zeros_postnan['Percent Zeros']
print(f'NUMBER OF VARIABLES AFTER Zeros THRESHOLD 6%: {df_test2.shape[1]}')
```

```
INITIAL NUMBER OF VARIABLES: 222

NUMBER OF VARIABLES AFTER NaN THRESHOLD 6%: 122

NUMBER OF VARIABLES AFTER Zeros THRESHOLD 6%: 62
```

In [70]:
```
# Check our filtered dataset
df_test2.describe()
```

Out[70]:

	Revenue	Revenue Growth	Gross Profit	SG&A Expense	Operati Expens
count	4.346000e+03	4253.000000	4.328000e+03	4.226000e+03	4.208000e+
mean	5.119287e+09	3.455278	2.043954e+09	9.005022e+08	1.435546e+
std	2.049504e+10	195.504906	7.682369e+09	3.661116e+09	5.529831e+
min	-6.894100e+07	-3.461500	-1.818220e+09	-1.401594e+08	-4.280000e+
25%	6.501425e+07	0.000000	3.618903e+07	2.056226e+07	4.223644e+
50%	4.982640e+08	0.074900	2.219470e+08	9.390450e+07	1.806253e+
75%	2.457878e+09	0.188500	9.767015e+08	4.117162e+08	6.796040e+
max	5.003430e+11	12739.000000	1.269470e+11	1.065100e+11	1.065100e+

8 rows × 61 columns

In [71]:
```
# Filter numeric columns
numeric_columns = df_test2.select_dtypes(include='number').columns

# Cut outliers
top_quantiles = df_test2[numeric_columns].quantile(0.97)
outliers_top = (df_test2[numeric_columns] > top_quantiles)
```

```
low_quantiles = df_test2[numeric_columns].quantile(0.03)
outliers_low = (df_test2[numeric_columns] < low_quantiles)

df_test2[numeric_columns] = df_test2[numeric_columns].mask(outliers_top, top
df_test2[numeric_columns] = df_test2[numeric_columns].mask(outliers_low, low

# Take a look at the dataframe post-outliers cut
df_test2.describe()
```

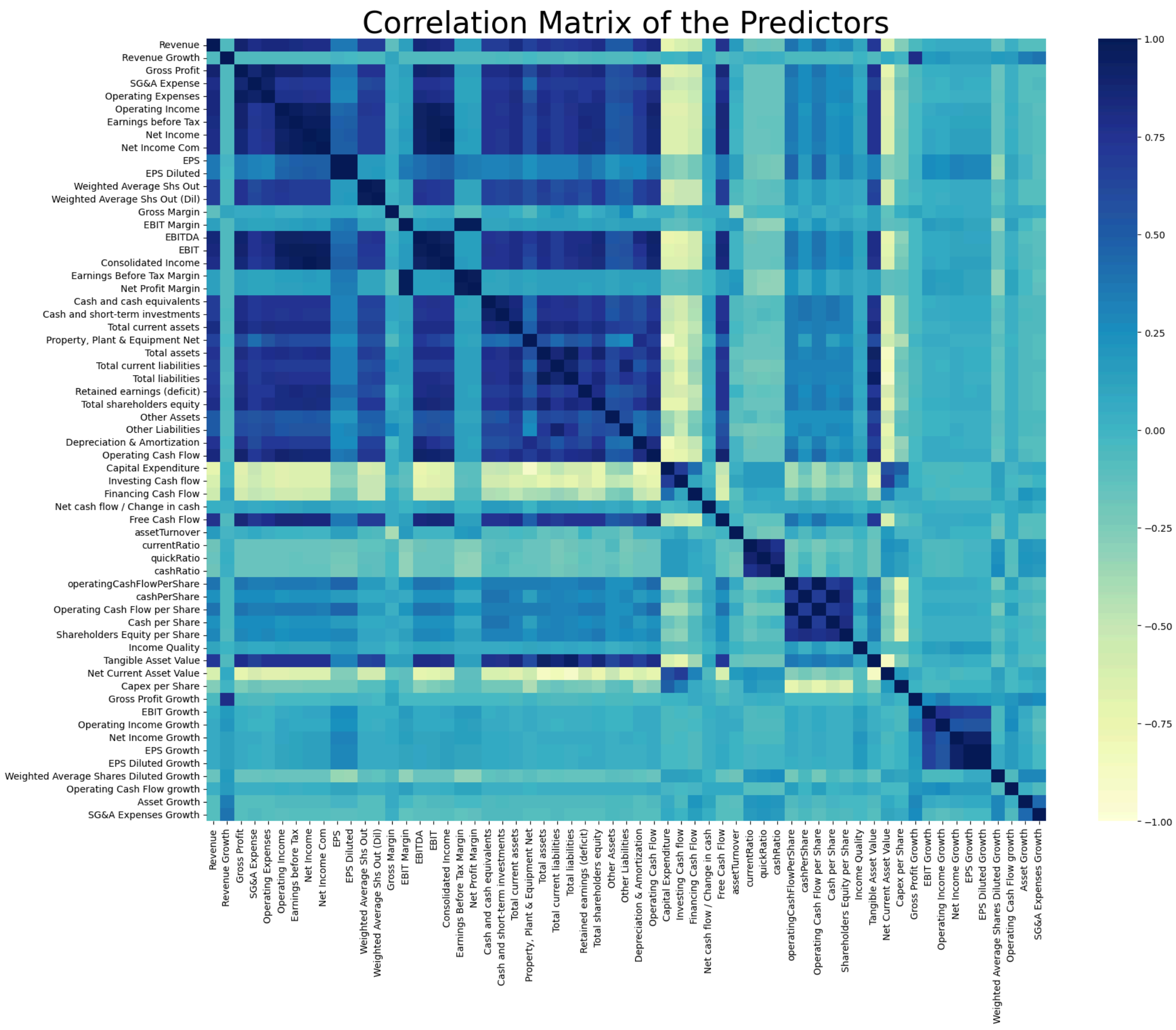
Out[71]:

	Revenue	Revenue Growth	Gross Profit	SG&A Expense	Operating Expenses
count	4.346000e+03	4253.000000	4.328000e+03	4.226000e+03	4.208000e+03
mean	3.437039e+09	0.135876	1.429547e+09	6.077564e+08	9.843496e+08
std	7.342150e+09	0.303442	3.264442e+09	1.394325e+09	2.222997e+09
min	0.000000e+00	-0.409488	0.000000e+00	8.908962e+05	4.198818e+06
25%	6.501425e+07	0.000000	3.618903e+07	2.056226e+07	4.223644e+07
50%	4.982640e+08	0.074900	2.219470e+08	9.390450e+07	1.806253e+08
75%	2.457878e+09	0.188500	9.767015e+08	4.117162e+08	6.796040e+08
max	3.366963e+10	1.248900	1.596702e+10	6.754875e+09	1.091602e+10

8 rows × 61 columns

In [72]:
```
# Replace nan-values with mean value of column's sector
for column in df_test2.select_dtypes(include=[np.number]).columns:
    df_test2[column] = df_test2.groupby('Sector')[column].transform(lambda x
```

In [73]:
```
# Plot correlation matrix of output dataset
fig, ax = plt.subplots(figsize=(20,15))
sns.heatmap(df_test2.select_dtypes(include=[np.number]).corr(), annot=False,
plt.title("Correlation Matrix of the Predictors", fontsize=32)
plt.show()
```



We outputted a correlation matrix of the indicators left, just to give a better sense of the relationships between the various indicators.

```
In [74]: # Add the sector column
df_out = df_test2.join(df['Sector'], rsuffix='_right')

# Add back the classification columns
df_out = df_out.join(class_data)

# Print information about dataset
df_out.info()
df_out.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4392 entries, CMCSA to ZYME
Data columns (total 65 columns):
 #   Column                                   Non-Null Count  Dtype  
---  ------                                   --------------  -----  
 0   Revenue                                  4392 non-null   float64
 1   Revenue Growth                           4392 non-null   float64
 2   Gross Profit                             4392 non-null   float64
 3   SG&A Expense                             4392 non-null   float64
 4   Operating Expenses                       4392 non-null   float64
 5   Operating Income                         4392 non-null   float64
 6   Earnings before Tax                      4392 non-null   float64
 7   Net Income                               4392 non-null   float64
 8   Net Income Com                           4392 non-null   float64
 9   EPS                                      4392 non-null   float64
 10  EPS Diluted                              4392 non-null   float64
 11  Weighted Average Shs Out                 4392 non-null   float64
 12  Weighted Average Shs Out (Dil)           4392 non-null   float64
 13  Gross Margin                             4392 non-null   float64
 14  EBIT Margin                              4392 non-null   float64
 15  EBITDA                                   4392 non-null   float64
 16  EBIT                                     4392 non-null   float64
 17  Consolidated Income                      4392 non-null   float64
 18  Earnings Before Tax Margin               4392 non-null   float64
 19  Net Profit Margin                        4392 non-null   float64
 20  Cash and cash equivalents                4392 non-null   float64
 21  Cash and short-term investments          4392 non-null   float64
 22  Total current assets                     4392 non-null   float64
 23  Property, Plant & Equipment Net          4392 non-null   float64
 24  Total assets                             4392 non-null   float64
 25  Total current liabilities                4392 non-null   float64
 26  Total liabilities                        4392 non-null   float64
 27  Retained earnings (deficit)              4392 non-null   float64
 28  Total shareholders equity                4392 non-null   float64
 29  Other Assets                             4392 non-null   float64
 30  Other Liabilities                        4392 non-null   float64
 31  Depreciation & Amortization              4392 non-null   float64
 32  Operating Cash Flow                      4392 non-null   float64
 33  Capital Expenditure                      4392 non-null   float64
 34  Investing Cash flow                      4392 non-null   float64
 35  Financing Cash Flow                      4392 non-null   float64
 36  Net cash flow / Change in cash           4392 non-null   float64
 37  Free Cash Flow                           4392 non-null   float64
 38  assetTurnover                            4392 non-null   float64
 39  currentRatio                             4392 non-null   float64
 40  quickRatio                               4392 non-null   float64
 41  cashRatio                                4392 non-null   float64
 42  operatingCashFlowPerShare                4392 non-null   float64
 43  cashPerShare                             4392 non-null   float64
 44  Operating Cash Flow per Share            4392 non-null   float64
 45  Cash per Share                           4392 non-null   float64
 46  Shareholders Equity per Share            4392 non-null   float64
 47  Income Quality                           4392 non-null   float64
 48  Tangible Asset Value                     4392 non-null   float64
 49  Net Current Asset Value                  4392 non-null   float64
 50  Capex per Share                          4392 non-null   float64
 51  Gross Profit Growth                          4392 non-null   float64
 52  EBIT Growth                                  4392 non-null   float64
 53  Operating Income Growth                      4392 non-null   float64
 54  Net Income Growth                            4392 non-null   float64
 55  EPS Growth                                   4392 non-null   float64
 56  EPS Diluted Growth                           4392 non-null   float64
 57  Weighted Average Shares Diluted Growth       4392 non-null   float64
 58  Operating Cash Flow growth                   4392 non-null   float64
 59  Asset Growth                                 4392 non-null   float64
 60  SG&A Expenses Growth                         4392 non-null   float64
 61  Sector                                       4392 non-null   object
 62  Sector_right                                 4392 non-null   object
 63  Class                                        4392 non-null   int64
 64  2019 PRICE VAR [%]                           4392 non-null   float64
dtypes: float64(62), int64(1), object(2)
memory usage: 2.3+ MB
```

Out[74]:

	Revenue	Revenue Growth	Gross Profit	SG&A Expense	Operating Expenses
count	4.392000e+03	4392.000000	4.392000e+03	4.392000e+03	4.392000e+03
mean	3.431959e+09	0.135713	1.429657e+09	6.163775e+08	9.916391e+08
std	7.304752e+09	0.298810	3.241932e+09	1.368804e+09	2.177880e+09
min	0.000000e+00	-0.409488	0.000000e+00	8.908962e+05	4.198818e+06
25%	6.584545e+07	0.000000	3.737700e+07	2.170000e+07	4.624375e+07
50%	5.200504e+08	0.078492	2.384675e+08	1.040695e+08	2.017610e+08
75%	2.577958e+09	0.184950	1.025554e+09	5.037902e+08	7.931652e+08
max	3.366963e+10	1.248900	1.596702e+10	6.754875e+09	1.091602e+10

8 rows × 63 columns

In [75]:

```
print(df_out.columns)
print(len(df_out.columns))
```

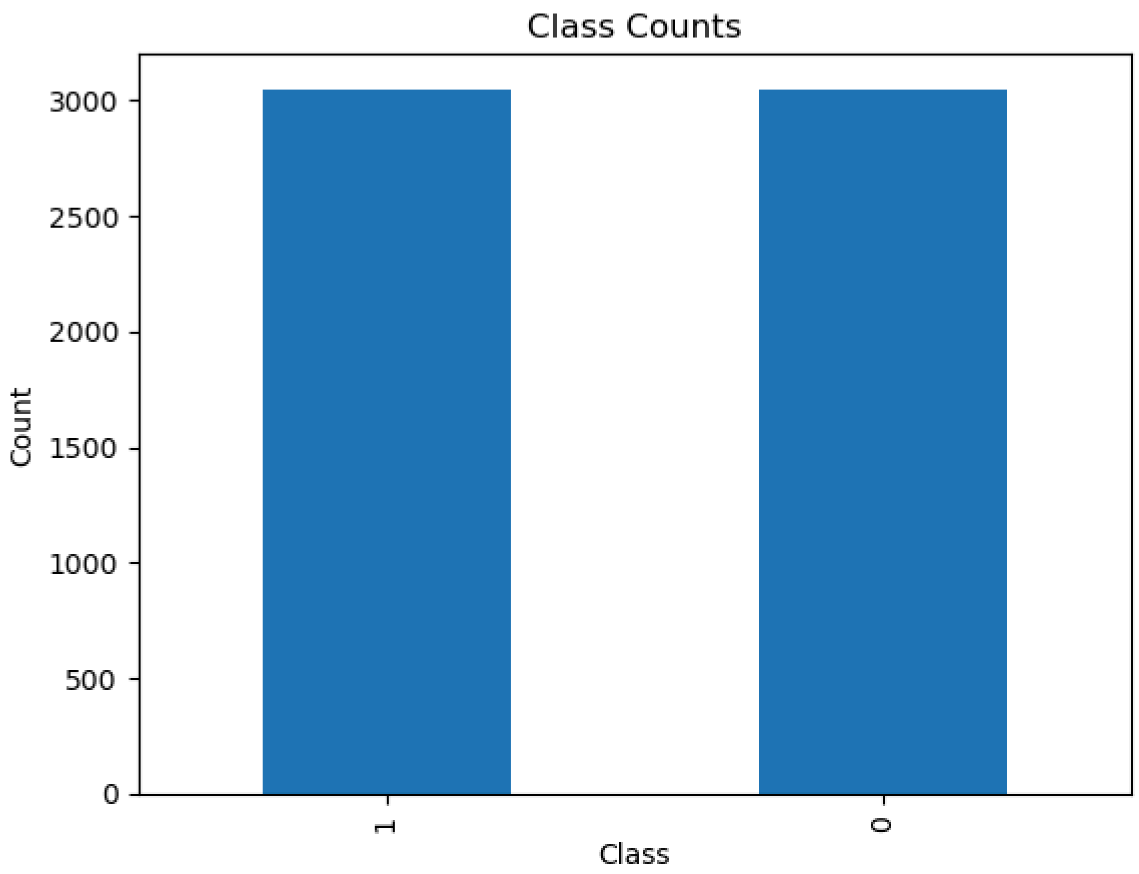
```
Index(['Revenue', 'Revenue Growth', 'Gross Profit', 'SG&A Expense',
       'Operating Expenses', 'Operating Income', 'Earnings before Tax',
       'Net Income', 'Net Income Com', 'EPS', 'EPS Diluted',
       'Weighted Average Shs Out', 'Weighted Average Shs Out (Dil)',
       'Gross Margin', 'EBIT Margin', 'EBITDA', 'EBIT', 'Consolidated Incom
e',
       'Earnings Before Tax Margin', 'Net Profit Margin',
       'Cash and cash equivalents', 'Cash and short-term investments',
       'Total current assets', 'Property, Plant & Equipment Net',
       'Total assets', 'Total current liabilities', 'Total liabilities',
       'Retained earnings (deficit)', 'Total shareholders equity',
       'Other Assets', 'Other Liabilities', 'Depreciation & Amortization',
       'Operating Cash Flow', 'Capital Expenditure', 'Investing Cash flow',
       'Financing Cash Flow', 'Net cash flow / Change in cash',
       'Free Cash Flow', 'assetTurnover', 'currentRatio', 'quickRatio',
       'cashRatio', 'operatingCashFlowPerShare', 'cashPerShare',
       'Operating Cash Flow per Share', 'Cash per Share',
       'Shareholders Equity per Share', 'Income Quality',
       'Tangible Asset Value', 'Net Current Asset Value', 'Capex per Share',
       'Gross Profit Growth', 'EBIT Growth', 'Operating Income Growth',
       'Net Income Growth', 'EPS Growth', 'EPS Diluted Growth',
       'Weighted Average Shares Diluted Growth', 'Operating Cash Flow growt
h',
       'Asset Growth', 'SG&A Expenses Growth', 'Sector', 'Sector_right',
       'Class', '2019 PRICE VAR [%]'],
      dtype='object')
65
```

Here, we use SMOTE to oversample and balance our classes --- SMOTE is an improved alternative to oversampling, which finds points that are closer in feature space, drawing a line between these points, and generating new data points along the line. We see from the plot that this new dataset, 'df_out_balanced', is balanced now. We will also run all of our models with the original dataset, in case SMOTE introduces too much noise.

In [76]:
```python
df_out2 = df_out.drop(['Sector', 'Sector_right'], axis=1)
```

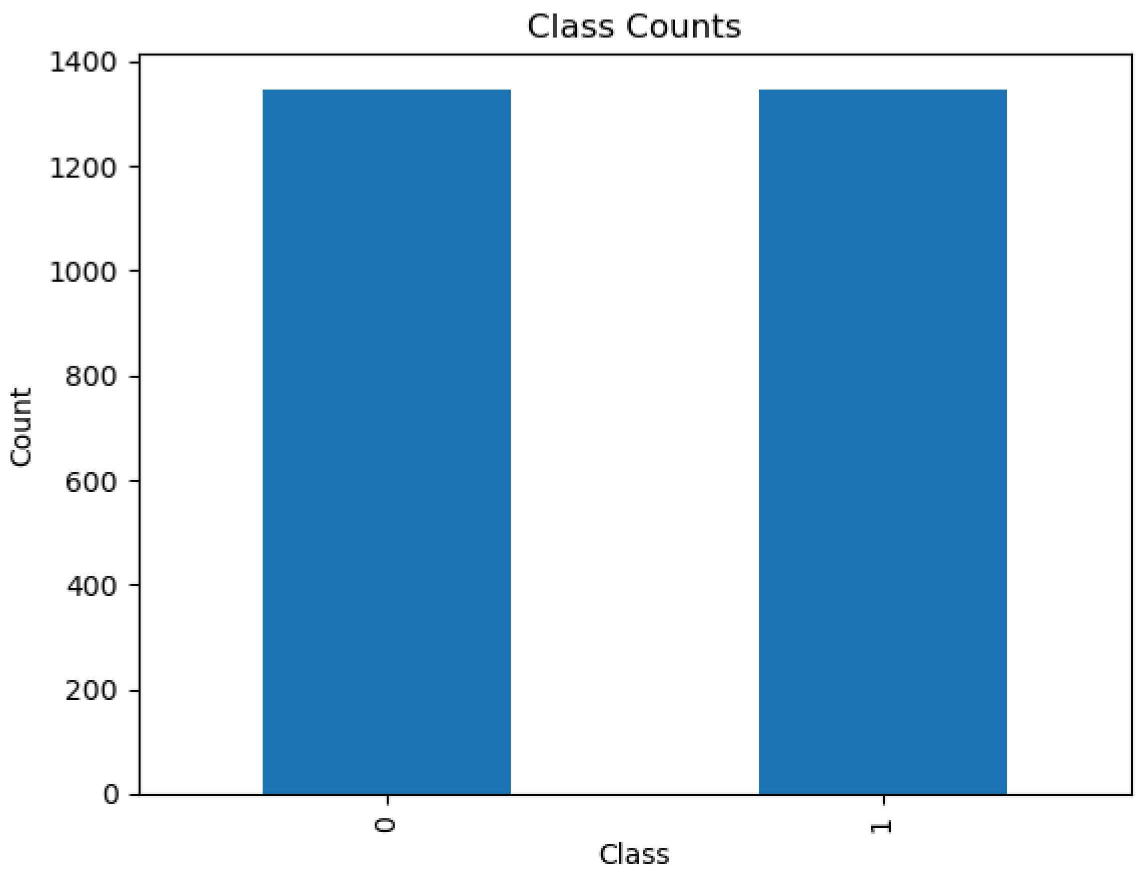
In [77]:
```python
# SMOTE oversampling
sm = SMOTE(random_state=42)
X_res, y_res = sm.fit_resample(df_out2.drop('Class', axis=1), df_out2['Class
df_out2 = pd.concat([pd.DataFrame(X_res), pd.DataFrame(y_res, columns=['Clas
```

In [78]:
```python
df_out2['Class'].value_counts().plot(kind='bar')
plt.title('Class Counts')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```



```
In [79]: # Undersampling
         rus = RandomUnderSampler(random_state=42)
         X_res, y_res = rus.fit_resample(df_out.drop('Class', axis=1), df_out['Class'
         df_out3 = pd.concat([pd.DataFrame(X_res), pd.DataFrame(y_res, columns=['Clas
```

```
In [80]: df_out3['Class'].value_counts().plot(kind='bar')
         plt.title('Class Counts')
         plt.xlabel('Class')
         plt.ylabel('Count')
         plt.show()
```



```
In [81]: df_out3 = df_out3.drop(['Sector', 'Sector_right'], axis=1)
```

```
In [82]: print(df_out.shape)
         print(df_out2.shape)
         print(df_out3.shape)
```

```
(4392, 65)
(6092, 63)
(2692, 63)
```

Part I: Basic Models

Return to contents

The main modelling question we are trying to solve is predicting the classification of a stock with its financial indicators (aka whether or not a stock price increased or decreased). First, we will do feature selection, and then we test 2 main basic models, knn/logistic regression and decision trees. Within these models, we will do fine-tuning as well as use various methods to identify feature importance.

Here is our model pipeline for our basic models:

Model Pipeline

Feature Selection

Return to contents

After taking care of data missingness and outliers, we are left with 63 columns, out of which 61 are financial indicators (and thus predictors). However, many of these predictors are highly correlated, so we further cut the number of predictors using feature selection, in all three of our dataframes (undersampling, oversampling, and original). To do this, we grouped features by features that are correlated by more than 0.90, and kept only one feature out of each of these groups to reduce redundancy. This left us with 41 predictors.

In [83]:

```
# Select only numeric columns for correlation matrix
numeric_df_out = df_out.select_dtypes(include=[np.number])
numeric_df_out2 = df_out2.select_dtypes(include=[np.number])
numeric_df_out3 = df_out3.select_dtypes(include=[np.number])

# Create correlation matrix for each dataframe
correlation_matrix = numeric_df_out.corr().abs()
correlation_matrix2 = numeric_df_out2.corr().abs()
correlation_matrix3 = numeric_df_out3.corr().abs()

# Select upper triangle of correlation matrix
upper = correlation_matrix.where(np.triu(np.ones(correlation_matrix.shape),
upper2 = correlation_matrix2.where(np.triu(np.ones(correlation_matrix2.shape
upper3 = correlation_matrix3.where(np.triu(np.ones(correlation_matrix3.shape

# Find index of feature columns with correlation greater than 0.90
to_drop = [column for column in upper.columns if any(upper[column] > 0.90)]
to_drop2 = [column for column in upper2.columns if any(upper2[column] > 0.9
to_drop3 = [column for column in upper3.columns if any(upper3[column] > 0.9

# Keep one feature from each group of highly correlated features
for group in to_drop:
    if isinstance(group, list):
        group.remove(group[0])
for group in to_drop2:
    if isinstance(group, list):
        group.remove(group[0])
for group in to_drop3:
    if isinstance(group, list):
        group.remove(group[0])

# Drop the remaining highly correlated features from each dataframe
df_out = df_out.drop(df_out[to_drop], axis=1)
df_out2 = df_out2.drop(df_out2[to_drop2], axis=1)
df_out3 = df_out3.drop(df_out3[to_drop3], axis=1)
```

```
In [84]: # For original dataframe:
         for column in to_drop:
             correlated = upper[upper[column] > 0.90].index.tolist()
             print(f"For df_out, dropped {column}, which was highly correlated with:
```

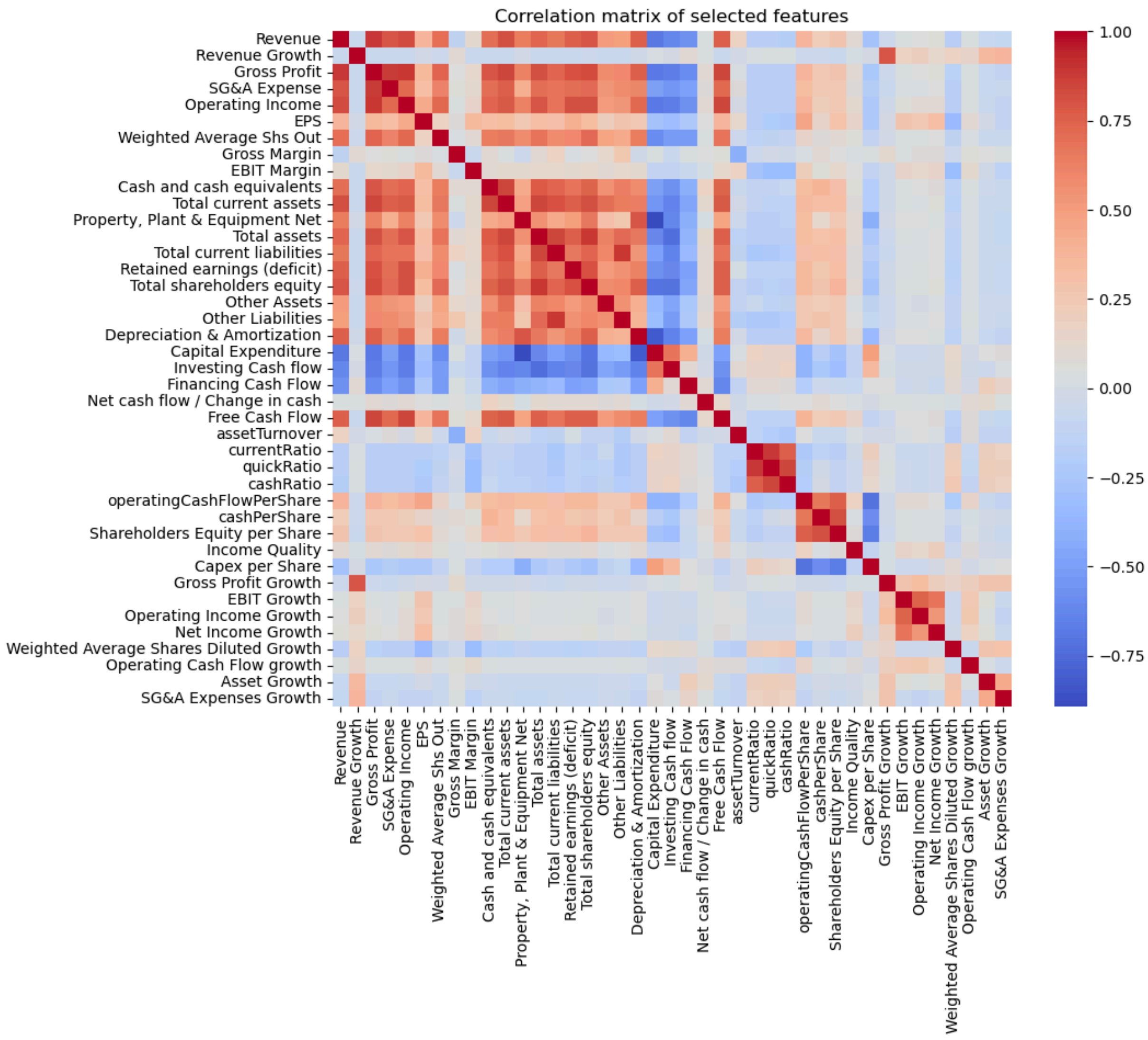
```
For df_out, dropped Operating Expenses, which was highly correlated with:
['Gross Profit', 'SG&A Expense']
For df_out, dropped Earnings before Tax, which was highly correlated with:
['Operating Income']
For df_out, dropped Net Income, which was highly correlated with: ['Operatin
g Income', 'Earnings before Tax']
For df_out, dropped Net Income Com, which was highly correlated with: ['Oper
ating Income', 'Earnings before Tax', 'Net Income']
For df_out, dropped EPS Diluted, which was highly correlated with: ['EPS']
For df_out, dropped Weighted Average Shs Out (Dil), which was highly correla
ted with: ['Weighted Average Shs Out']
For df_out, dropped EBITDA, which was highly correlated with: ['Gross Profi
t', 'Operating Income', 'Earnings before Tax', 'Net Income', 'Net Income Co
m']
For df_out, dropped EBIT, which was highly correlated with: ['Operating Inco
me', 'Earnings before Tax', 'Net Income', 'Net Income Com', 'EBITDA']
For df_out, dropped Consolidated Income, which was highly correlated with:
['Operating Income', 'Earnings before Tax', 'Net Income', 'Net Income Com',
'EBITDA', 'EBIT']
For df_out, dropped Earnings Before Tax Margin, which was highly correlated
with: ['EBIT Margin']
For df_out, dropped Net Profit Margin, which was highly correlated with: ['E
BIT Margin', 'Earnings Before Tax Margin']
For df_out, dropped Cash and short-term investments, which was highly correl
ated with: ['Cash and cash equivalents']
For df_out, dropped Total liabilities, which was highly correlated with: ['T
otal assets']
For df_out, dropped Operating Cash Flow, which was highly correlated with:
['EBITDA', 'EBIT']
For df_out, dropped Operating Cash Flow per Share, which was highly correlat
ed with: ['operatingCashFlowPerShare']
For df_out, dropped Cash per Share, which was highly correlated with: ['cash
PerShare']
For df_out, dropped Tangible Asset Value, which was highly correlated with:
['Total assets', 'Total liabilities']
For df_out, dropped Net Current Asset Value, which was highly correlated wit
h: ['Total liabilities']
For df_out, dropped EPS Growth, which was highly correlated with: ['Net Inco
me Growth']
For df_out, dropped EPS Diluted Growth, which was highly correlated with:
['Net Income Growth', 'EPS Growth']
```

```
In [85]: print(df_out.columns)
         print(len(df_out.columns))
```

```
Index(['Revenue', 'Revenue Growth', 'Gross Profit', 'SG&A Expense',
       'Operating Income', 'EPS', 'Weighted Average Shs Out', 'Gross Margi
n',
       'EBIT Margin', 'Cash and cash equivalents', 'Total current assets',
       'Property, Plant & Equipment Net', 'Total assets',
       'Total current liabilities', 'Retained earnings (deficit)',
       'Total shareholders equity', 'Other Assets', 'Other Liabilities',
       'Depreciation & Amortization', 'Capital Expenditure',
       'Investing Cash flow', 'Financing Cash Flow',
       'Net cash flow / Change in cash', 'Free Cash Flow', 'assetTurnover',
       'currentRatio', 'quickRatio', 'cashRatio', 'operatingCashFlowPerShar
e',
       'cashPerShare', 'Shareholders Equity per Share', 'Income Quality',
       'Capex per Share', 'Gross Profit Growth', 'EBIT Growth',
       'Operating Income Growth', 'Net Income Growth',
       'Weighted Average Shares Diluted Growth', 'Operating Cash Flow growt
h',
       'Asset Growth', 'SG&A Expenses Growth', 'Sector', 'Sector_right',
       'Class', '2019 PRICE VAR [%]'],
      dtype='object')
45
```

In [86]:
```
#features = ['Revenue', 'Revenue Growth', 'Operating Income', 'EPS', 'Gross
features = [col for col in df_out.columns if col not in ['Sector', 'Sector_r
```

In [87]:
```
correlation_matrix = df_out[features].corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm')
plt.title('Correlation matrix of selected features')
plt.show()
```



We visualized the correlations between the leftover features once more, to see if any features are still highly correlated. Although there are some that are still pretty correlated, we will keep all of these features, as the discrepancies might still be important.

In [88]:
```
selected_features_df = df_out[features]
selected_features_df.head()
```

Out[88]:

	Revenue	Revenue Growth	Gross Profit	SG&A Expense	Operating Income	
CMCSA	3.366963e+10	0.1115	1.596702e+10	6.754875e+09	5.184200e+09	
KMI	1.414400e+10	0.0320	6.856000e+09	6.010000e+08	3.794000e+09	
INTC	3.366963e+10	0.1289	1.596702e+10	6.750000e+09	5.184200e+09	
MU	3.039100e+10	0.4955	1.596702e+10	8.130000e+08	5.184200e+09	
GE	3.366963e+10	0.0285	1.596702e+10	6.754875e+09	-1.799788e+08	-

5 rows × 41 columns

K-nn & Logistic Regression

Return to contents

First, let us fit a logistic regression model to predict whether a stock will increase or decrease from revenue alone, just as a baseline model. We will also fit this model on all three datasets we have --- the original one, the SMOTE one, and the undersampled one, for comparison.

In [89]:
```
X = df_out[features]
y = df_out['Class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar
```

In [90]:
```
# Fit the logistic regression model
logit1 = LogisticRegression(max_iter=1000)
logit1.fit(X_train[['Revenue']], y_train)

# Store the learned parameters
logit1_beta0 = logit1.intercept_[0]
logit1_beta1 = logit1.coef_[0][0]

# Predict on the train and test sets
y_train_pred = logit1.predict(X_train[['Revenue']])
y_test_pred = logit1.predict(X_test[['Revenue']])

# Calculate and store the train and test classification accuracies
acc_train_logit1 = accuracy_score(y_train, y_train_pred)
acc_test_logit1 = accuracy_score(y_test, y_test_pred)
```

In [91]:
```
print("LOGISTIC REGRESSION FOR ORIGINAL DATASET")
print("Learned Parameters:")
print("Intercept: ", logit1_beta0)
print("Coefficient: ", logit1_beta1)
print("\nClassification Accuracies:")
print("Train Accuracy: ", acc_train_logit1)
print("Test Accuracy: ", acc_test_logit1)
```

```
LOGISTIC REGRESSION FOR ORIGINAL DATASET
Learned Parameters:
Intercept:  2.6749427288409396e-19
Coefficient:  1.1517404119857235e-10

Classification Accuracies:
Train Accuracy:  0.6945630515229149
Test Accuracy:  0.689419795221843
```

In [92]:
```
features_without_capex_quickratio = [feature for feature in features if feat
X2 = df_out2[features_without_capex_quickratio]
y2 = df_out2['Class']
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y2, test_size=0.

# Fit the logistic regression model
logit1_2 = LogisticRegression(max_iter=1000)
logit1_2.fit(X_train2[['Revenue']], y_train2)

# Store the learned parameters
logit1_beta0_2 = logit1_2.intercept_[0]
logit1_beta1_2 = logit1_2.coef_[0][0]

# Predict on the train and test sets
y_train_pred2 = logit1_2.predict(X_train2[['Revenue']])
y_test_pred2 = logit1_2.predict(X_test2[['Revenue']])

# Calculate and store the train and test classification accuracies
acc_train_logit1_2 = accuracy_score(y_train2, y_train_pred2)
acc_test_logit1_2 = accuracy_score(y_test2, y_test_pred2)

print("LOGISTIC REGRESSION FOR OVERSAMPLED DATASET")
print("Learned Parameters:")
print("Intercept: ", logit1_beta0_2)
print("Coefficient: ", logit1_beta1_2)
print("\nClassification Accuracies:")
print("Train Accuracy: ", acc_train_logit1_2)
print("Test Accuracy: ", acc_test_logit1_2)
```

```
LOGISTIC REGRESSION FOR OVERSAMPLED DATASET
Learned Parameters:
Intercept:  -1.2284368840404984e-20
Coefficient:  4.526647260651638e-11

Classification Accuracies:
Train Accuracy:  0.5005130309870717
Test Accuracy:  0.5365053322395406
```

In [93]:
```
features_without_unwanted = [feature for feature in features if feature not
X3 = df_out3[features_without_unwanted]
y3 = df_out3['Class']
X_train3, X_test3, y_train3, y_test3 = train_test_split(X3, y3, test_size=0.

# Fit the logistic regression model
logit1_3 = LogisticRegression(max_iter=1000)
```

```
logit1_3.fit(X_train3[['Revenue']], y_train3)

# Store the learned parameters
logit1_beta0_3 = logit1_3.intercept_[0]
logit1_beta1_3 = logit1_3.coef_[0][0]

# Predict on the train and test sets
y_train_pred3 = logit1_3.predict(X_train3[['Revenue']])
y_test_pred3 = logit1_3.predict(X_test3[['Revenue']])

# Calculate and store the train and test classification accuracies
acc_train_logit1_3 = accuracy_score(y_train3, y_train_pred3)
acc_test_logit1_3 = accuracy_score(y_test3, y_test_pred3)

print("LOGISTIC REGRESSION FOR UNDERSAMPLED DATASET")
print("Learned Parameters:")
print("Intercept: ", logit1_beta0_3)
print("Coefficient: ", logit1_beta1_3)
print("\nClassification Accuracies:")
print("Train Accuracy: ", acc_train_logit1_3)
print("Test Accuracy: ", acc_test_logit1_3)
```

```
LOGISTIC REGRESSION FOR UNDERSAMPLED DATASET
Learned Parameters:
Intercept:  -1.2233649397135117e-20
Coefficient:  4.720783021288649e-11

Classification Accuracies:
Train Accuracy:  0.5290292614955876
Test Accuracy:  0.5009276437847866
```

In [94]:
```
classification_accuracies = pd.DataFrame({
    'Original': [acc_train_logit1, acc_test_logit1],
    'Oversampling (SMOTE)': [acc_train_logit1_2, acc_test_logit1_2],
    'Random Undersampling': [acc_train_logit1_3, acc_test_logit1_3]
}, index=['Train Accuracy', 'Test Accuracy'])

print(tabulate(classification_accuracies, headers='keys', tablefmt='psql'))
```

```
+----------------+------------+------------------------+--------------------
----+
|                |   Original |   Oversampling (SMOTE) |   Random Undersampl
ing |
|----------------+------------+------------------------+--------------------
----|
| Train Accuracy |   0.694563 |               0.500513 |               0.529
029 |
| Test Accuracy  |   0.68942  |               0.536505 |               0.500
928 |
+----------------+------------+------------------------+--------------------
----+
```

From these results, we see that both the train and test accuracy are much less accurate for the SMOTE and Random Undersampling dataset than the original dataset, indicating that the altered datasets might be introducing too much noise, or giving too little information. Since the test accuracy of the original model is also doing much better, we decided to not use the oversampled or undersampled dataset and stick with out original dataset.

Next, let us fit a logistic regression model predicting stock increase or decrease with just 'Revenue' and 'EPS'.

In [95]:
```
# Fit the logistic regression model with 'Revenue' and 'EPS'
logit2 = LogisticRegression(max_iter=1000)
logit2.fit(X_train[['Revenue', 'EPS']], y_train)

# Store the learned parameters
logit2_beta0 = logit2.intercept_[0]
logit2_beta1, logit2_beta2 = logit2.coef_[0]

# Predict on the train and test sets
y_train_pred2 = logit2.predict(X_train[['Revenue', 'EPS']])
y_test_pred2 = logit2.predict(X_test[['Revenue', 'EPS']])

# Calculate and store the train and test classification accuracies
acc_train_logit2 = accuracy_score(y_train, y_train_pred2)
acc_test_logit2 = accuracy_score(y_test, y_test_pred2)

print("Learned Parameters:")
print("Intercept: ", logit2_beta0)
print("Coefficient for Revenue: ", logit2_beta1)
print("Coefficient for Revenue Growth: ", logit2_beta2)
print("\nClassification Accuracies:")
print("Train Accuracy: ", acc_train_logit2)
print("Test Accuracy: ", acc_test_logit2)
```

```
Learned Parameters:
Intercept:  2.67494272884260466e-19
Coefficient for Revenue:  1.1517404119857249e-10
Coefficient for Revenue Growth:  7.1411856671072815e-19

Classification Accuracies:
Train Accuracy:  0.694847708511244
Test Accuracy:  0.7030716723549488
```

Next, let's fit a full logistic regression model predicting stock increase or decrease with all 16 features.

In [96]:
```
logit3 = LogisticRegression(max_iter=1000)
logit3.fit(X_train, y_train)

# Store the learned parameters
logit3_beta = logit3.coef_[0]
logit3_intercept = logit3.intercept_[0]

# Predict on the train and test sets
y_train_pred3 = logit3.predict(X_train)
y_test_pred3 = logit3.predict(X_test)

# Calculate and store the train and test classification accuracies
```

```
acc_train_logit3 = accuracy_score(y_train, y_train_pred3)
acc_test_logit3 = accuracy_score(y_test, y_test_pred3)

print("Learned Parameters:")
print("Intercept: ", logit3_intercept)
print("Coefficients: ", logit3_beta)
print("\nClassification Accuracies:")
print("Train Accuracy: ", acc_train_logit3)
print("Test Accuracy: ", acc_test_logit3)
```

```
Learned Parameters:
Intercept:  8.587470417885868e-17
Coefficients:  [ 2.11649912e-11  4.53800922e-18 -2.46046196e-10  2.65053659e
-10
  6.74247233e-10  2.81759925e-16  8.17598325e-10  5.25853156e-17
  7.56372164e-17  2.77939497e-11  4.60628217e-11  2.12933417e-11
 -2.07431601e-11  9.18128295e-12 -7.03577643e-11  8.61268734e-11
 -2.55722700e-12  2.65898432e-10  3.52371679e-10  6.09701599e-10
 -2.34592552e-10  3.66955255e-10  1.07795957e-10  1.86159317e-10
  4.37614167e-17  1.83470807e-16  1.28022630e-16  5.50320635e-17
  2.06314656e-16 -6.48810737e-17  1.09848096e-15  1.31386599e-16
 -6.34924401e-17  7.71723418e-18  2.59854321e-17  1.68422823e-17
  5.83378390e-17 -6.67853884e-18  2.06992884e-17  7.86767973e-18
  7.37740909e-18]

Classification Accuracies:
Train Accuracy:  0.6945630515229149
Test Accuracy:  0.6951080773606371
```

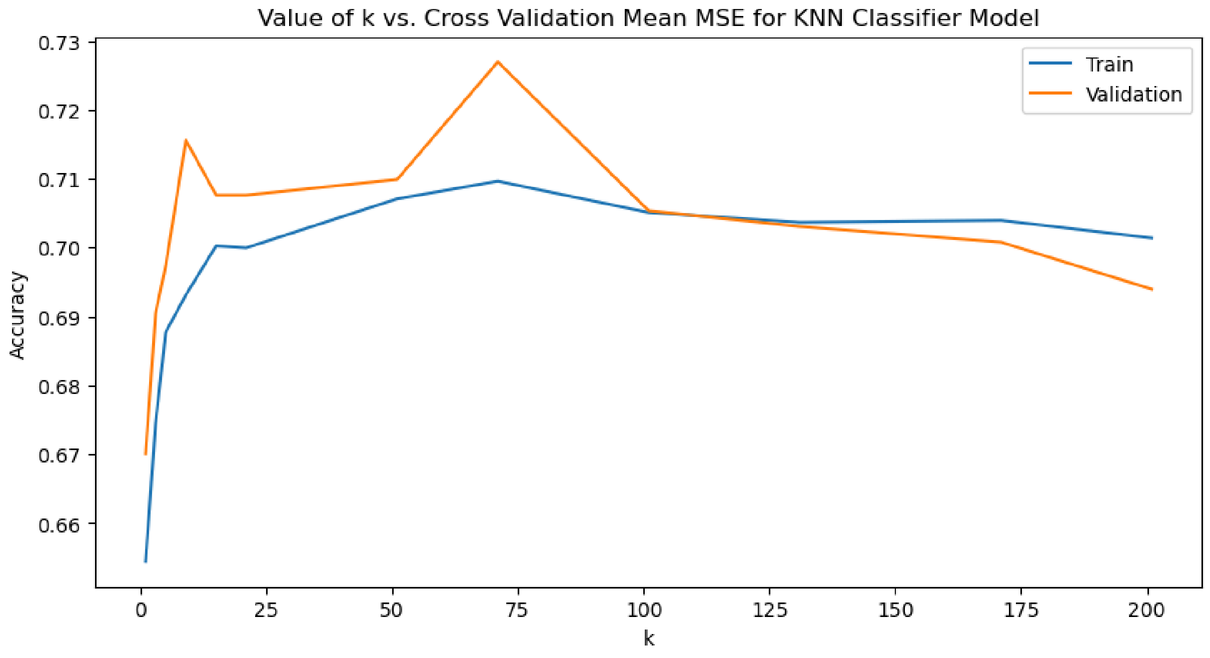
Next, we scaled our predictors, so that we can weigh our parameters similarly in future models. This also helps for features that have completely different scales, like revenue vs current ratio.

In [97]:
```
# Initialize a scaler
scaler = StandardScaler()

# Fit the scaler on the training data
scaler.fit(X_train)

# Transform both the train and test data
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Next, we fit a well-tuned k-NN classification model with main effects of all 16 predictors in it, using 10-fold cross-validation with classification accuracy as the scoring metric. After trying many k values, we plot the training and validation scores of the model at each value of k, and then chose the k with the validation accuracy.

In [98]:
```
ks = [1, 3, 5, 9, 15, 21, 51, 71, 101, 131, 171, 201]
mean_train_scores = []
mean_val_scores = []

for k in ks:
    knn_model = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn_model, X_train, y_train, cv=10, scoring='ac
    mean_train_scores.append(scores.mean())

    knn_model.fit(X_train, y_train)
    val_score = knn_model.score(X_test, y_test)
    mean_val_scores.append(val_score)
```

In [99]:
```
# Plotting the scores
plt.figure(figsize=(10, 5))
plt.plot(ks, mean_train_scores, label='Train')
plt.plot(ks, mean_val_scores, label='Validation')
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.title("Value of k vs. Cross Validation Mean MSE for KNN Classifier Model
plt.legend()
plt.show()
```



In [100…
```
# Storing the best k and the classification accuracy on train and test
best_k = ks[mean_val_scores.index(max(mean_val_scores))]
knn_train_acc = max(mean_train_scores)
knn_test_acc = max(mean_val_scores)
```

In [101…
```
print(f"Best k: {best_k}")
print(f"Train accuracy: {knn_train_acc}")
print(f"Test accuracy: {knn_test_acc}")
```

```
Best k: 71
Train accuracy: 0.709648569023569
Test accuracy: 0.726962457337884
```

Next, we train and test a full logistic regression model with all features included, with scaling.

In [102…

```python
logit4 = LogisticRegression(max_iter=1000)
logit4.fit(X_train, y_train)

# Store the learned parameters
logit4_beta = logit3.coef_[0]
logit4_intercept = logit3.intercept_[0]

# Predict on the train and test sets
y_train_pred4 = logit4.predict(X_train)
y_test_pred4 = logit4.predict(X_test)

# Calculate and store the train and test classification accuracies
acc_train_logit4 = accuracy_score(y_train, y_train_pred4)
acc_test_logit4 = accuracy_score(y_test, y_test_pred4)

print("Learned Parameters:")
print("Intercept: ", logit4_intercept)
print("Coefficients: ", logit4_beta)
print("\nClassification Accuracies:")
print("Train Accuracy: ", acc_train_logit4)
print("Test Accuracy: ", acc_test_logit4)
```

```
Learned Parameters:
Intercept:  8.587470417885868e-17
Coefficients:  [ 2.11649912e-11  4.53800922e-18 -2.46046196e-10  2.65053659e
-10
  6.74247233e-10  2.81759925e-16  8.17598325e-10  5.25853156e-17
  7.56372164e-17  2.77939497e-11  4.60628217e-11  2.12933417e-11
 -2.07431601e-11  9.18128295e-12 -7.03577643e-11  8.61268734e-11
 -2.55722700e-12  2.65898432e-10  3.52371679e-10  6.09701599e-10
 -2.34592552e-10  3.66955255e-10  1.07795957e-10  1.86159317e-10
  4.37614167e-17  1.83470807e-16  1.28022630e-16  5.50320635e-17
  2.06314656e-16 -6.48810737e-17  1.09848096e-15  1.31386599e-16
 -6.34924401e-17  7.71723418e-18  2.59854321e-17  1.68422823e-17
  5.83378390e-17 -6.67853884e-18  2.06992884e-17  7.86767973e-18
  7.37740909e-18]

Classification Accuracies:
Train Accuracy:  0.7147736976942783
Test Accuracy:  0.7087599544937428
```

In [103…

```python
# Initialize lists to store results and coefficients
logit_lasso_train_accs = []
logit_lasso_test_accs = []
logit_lasso_coefs = []

# Define the C values to try
Cs = [1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3, 1e4]

# Loop over the C values
for C in Cs:
    # Fit a Lasso-like logistic regression model
    logit_lasso = LogisticRegression(C=C, penalty='l1', solver='saga', max_i
    logit_lasso.fit(X_train, y_train)

    # Predict on the train and test sets
    y_train_pred_lasso = logit_lasso.predict(X_train)
    y_test_pred_lasso = logit_lasso.predict(X_test)

    # Calculate and store the train and test classification accuracies
    logit_lasso_train_acc = accuracy_score(y_train, y_train_pred_lasso)
    logit_lasso_test_acc = accuracy_score(y_test, y_test_pred_lasso)

    # Store the results and coefficients
    logit_lasso_train_accs.append(logit_lasso_train_acc)
    logit_lasso_test_accs.append(logit_lasso_test_acc)
    logit_lasso_coefs.append(logit_lasso.coef_)

# Find the index of the best test accuracy
best_index = np.argmax(logit_lasso_test_accs)
print("Best C value: ", Cs[best_index])


# Print the best results
print("Best Train Accuracy: ", logit_lasso_train_accs[best_index])
print("Best Test Accuracy: ", logit_lasso_test_accs[best_index])
print("Best Coefficients: ", logit_lasso_coefs[best_index])
```

```
Best C value:  0.1
Best Train Accuracy:  0.7136350697409621
Best Test Accuracy:  0.7178612059158134
Best Coefficients:  [[ 0.16745119 -0.08065373 -0.02658166  0.          0.
0.46664148
   0.          0.09336308  0.16518824  0.00273608  0.13278026  0.
   0.          0.         -0.05100065  0.          0.          0.27953223
   0.          0.0696631  -0.05518079  0.11362988  0.00343962  0.04548695
  -0.24943294 -0.06975951 -0.02966424  0.         -0.01525486 -0.14693989
   0.02731399  0.02032016  0.08170126  0.00806167 -0.05438753  0.
   0.         -0.15216239  0.          0.00968026  0.04806951]]
```

In [104…

```
# Create a DataFrame for non-zero coefficients
non_zero_coefs = pd.DataFrame({'Feature': features, 'Coefficient': logit_las
non_zero_coefs = non_zero_coefs[non_zero_coefs['Coefficient'] != 0]

# Sort the DataFrame by coefficient value in ascending order
non_zero_coefs = non_zero_coefs.sort_values('Coefficient', ascending=False)

print("NONZERO COEFFICIENTS")
# Print the DataFrame
print(non_zero_coefs)
```

```
NONZERO COEFFICIENTS
                                  Feature  Coefficient
5                                     EPS     0.466641
17                      Other Liabilities     0.279532
0                                 Revenue     0.167451
8                             EBIT Margin     0.165188
10                   Total current assets     0.132780
21                    Financing Cash Flow     0.113630
7                            Gross Margin     0.093363
32                        Capex per Share     0.081701
19                    Capital Expenditure     0.069663
40                   SG&A Expenses Growth     0.048070
23                         Free Cash Flow     0.045487
30           Shareholders Equity per Share     0.027314
31                         Income Quality     0.020320
39                           Asset Growth     0.009680
33                    Gross Profit Growth     0.008062
22            Net cash flow / Change in cash     0.003440
9               Cash and cash equivalents     0.002736
28              operatingCashFlowPerShare    -0.015255
2                            Gross Profit    -0.026582
26                             quickRatio    -0.029664
14             Retained earnings (deficit)    -0.051001
34                            EBIT Growth    -0.054388
20                    Investing Cash flow    -0.055181
25                           currentRatio    -0.069760
1                          Revenue Growth    -0.080654
29                           cashPerShare    -0.146940
37  Weighted Average Shares Diluted Growth    -0.152162
24                          assetTurnover    -0.249433
```

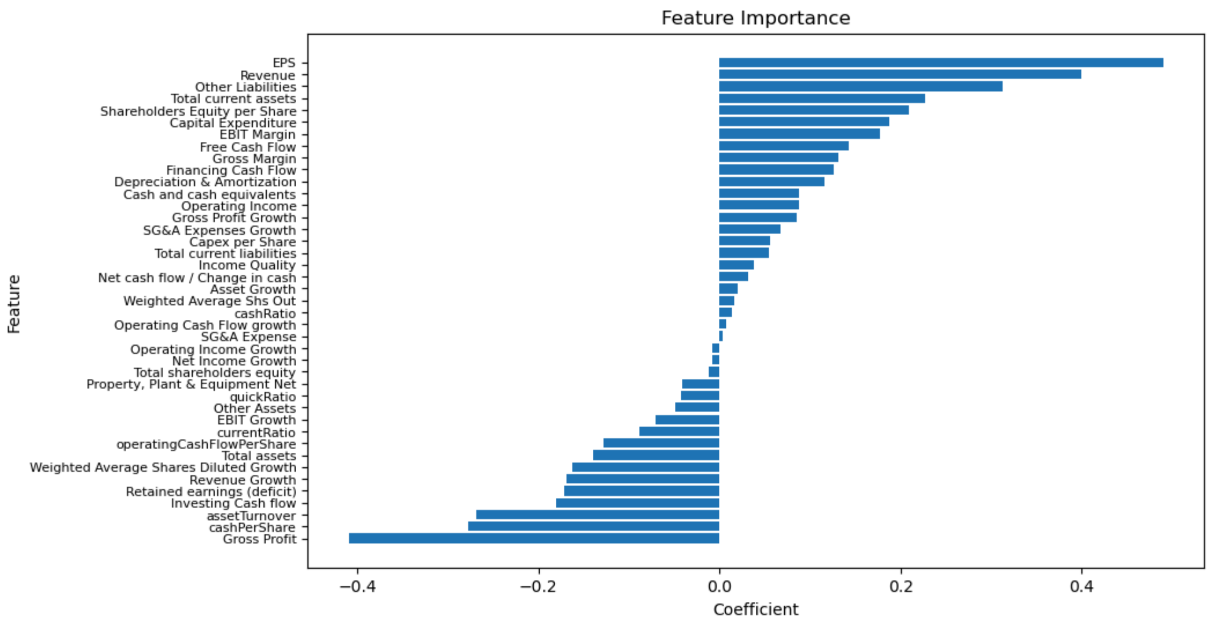
In [105…

```
# Create a DataFrame for zero coefficients
zero_coefs = pd.DataFrame({'Feature': features, 'Coefficient': logit_lasso_c
zero_coefs = zero_coefs[zero_coefs['Coefficient'] == 0]

# Sort the DataFrame by coefficient value in ascending order
zero_coefs = zero_coefs.sort_values('Coefficient', ascending=False)

print("ZERO COEFFICIENTS")
# Print the DataFrame
print(zero_coefs)
```

```
ZERO COEFFICIENTS
                            Feature  Coefficient
3                       SG&A Expense          0.0
4                   Operating Income          0.0
6           Weighted Average Shs Out          0.0
11  Property, Plant & Equipment Net          0.0
12                      Total assets          0.0
13         Total current liabilities          0.0
15         Total shareholders equity          0.0
16                      Other Assets          0.0
18       Depreciation & Amortization          0.0
27                         cashRatio          0.0
35           Operating Income Growth          0.0
36                 Net Income Growth          0.0
38       Operating Cash Flow growth          0.0
```

In [106…

```python
# Get coefficients from the logistic regression model
coefficients = logit_lasso.coef_[0]
coef_df = pd.DataFrame({'feature': features, 'coefficient': coefficients})
coef_df = coef_df.sort_values('coefficient', ascending=False)

# Plot the coefficients
plt.figure(figsize=(10, 6))
plt.barh(coef_df['feature'], coef_df['coefficient'])
plt.xlabel('Coefficient')
plt.ylabel('Feature')
plt.title('Feature Importance')
plt.gca().invert_yaxis()
plt.yticks(fontsize=8)  # Adjust the y-ticks to be slightly farther apart
plt.show()
```



In [107…

```python
# Store all the train and test accuracies as well as the model description i
model_results = pd.DataFrame({
    'Model': ['Logistic Regression (just Revenue)', 'Logistic Regression (Re
    'Train Accuracy': [acc_train_logit1, acc_train_logit2, acc_train_logit3,
    'Test Accuracy': [acc_test_logit1, acc_test_logit2, acc_test_logit3, acc
})
```

```
model_results
```

Out[107…

	Model	Train Accuracy	Test Accuracy
0	Logistic Regression (just Revenue)	0.694563	0.689420
1	Logistic Regression (Revenue + EPS)	0.694848	0.703072
2	Logistic Regression (all features, no scaling)	0.694563	0.695108
3	Logistic Regression (all features, scaled)	0.714774	0.708760
4	Logistic Regression with Lasso	0.713635	0.717861
5	KNN (k=71)	0.709649	0.726962

After training logistic regression models on various parameters, with scaled and unscaled features, and with lasso regression, we see that the best performing logistic regression models (based on test accuracy), is the logistic regression model on all scaled features with lasso regulation, with a test accuracy of 71.79%. We see that the logistic regression model based on just revenue and EPS and the logistic regression model based on all scaled features also did well, with test accuracies of 70.31% and 70.87%, respectively.

From the lasso regularization, 13 predictors came out with coefficients of 0, including Operating Income, Total current liabilities, and Operating Income Growth, which indicates that they might be less signigicant as predictors of stock increase or decrease than other features. Additionally, from plotting the coefficients of both the logistic regression and the logistic regression with lasso, we see that EPS and Revenue are the two features with the greatest coefficients, indicating that an increase in EPS or Revenue increases the odds of a stock increasing the most. This generally makes sense, as EPS (Earnings Per Share) is a direct measure of a company's profitability on a per-share basis, and high EPS can incease investor confidence in a company, thus leading to increase in stock prices. Revenue, as the first raw measure of how much money is making, also makes sense as a strong positive predictor of a stock's increase, as revenue is not only a easily accessible comparison metric, but also indicator of business growth and investor confidence. However, a more interesting result is that Gross Profit came out as the strongest predictor of stock price decrease, which doesn't make too much intuitive sense, since Gross Profit is another indicator of a business' growth and earnings. However, there could be some randomness involved in this, or other context --- for example, increase in gross profit could come from one-time events or non-operational activities like selling assets, which could negatively impact stock price.

We also ran one k-nn model on the features, choosing the k that yielded the highest validation score, which was k=71. This k-nn model actually performed better than all of the logistic regression models in terms of test accuracy, which may be because k-nn models can capture complex decision boundaries, which is relevant here because of the number of features we have. However, it was still important to run the logistic regression model, as it has more interpretable coefficients that can tell us the predicting power of each feature.

Decision Trees

Return to contents

Next, we evaluate and tune a decision tree classifier along with regularization methods like bagging, random forest, and boosting. This provides an alternative from logistic regression that can capture complex decision boundaries, which may be useful with the complexity of our features and data.

First, we fit a decision tree classifier to the data with 20 different depths, choosing the best performing depth as our result.

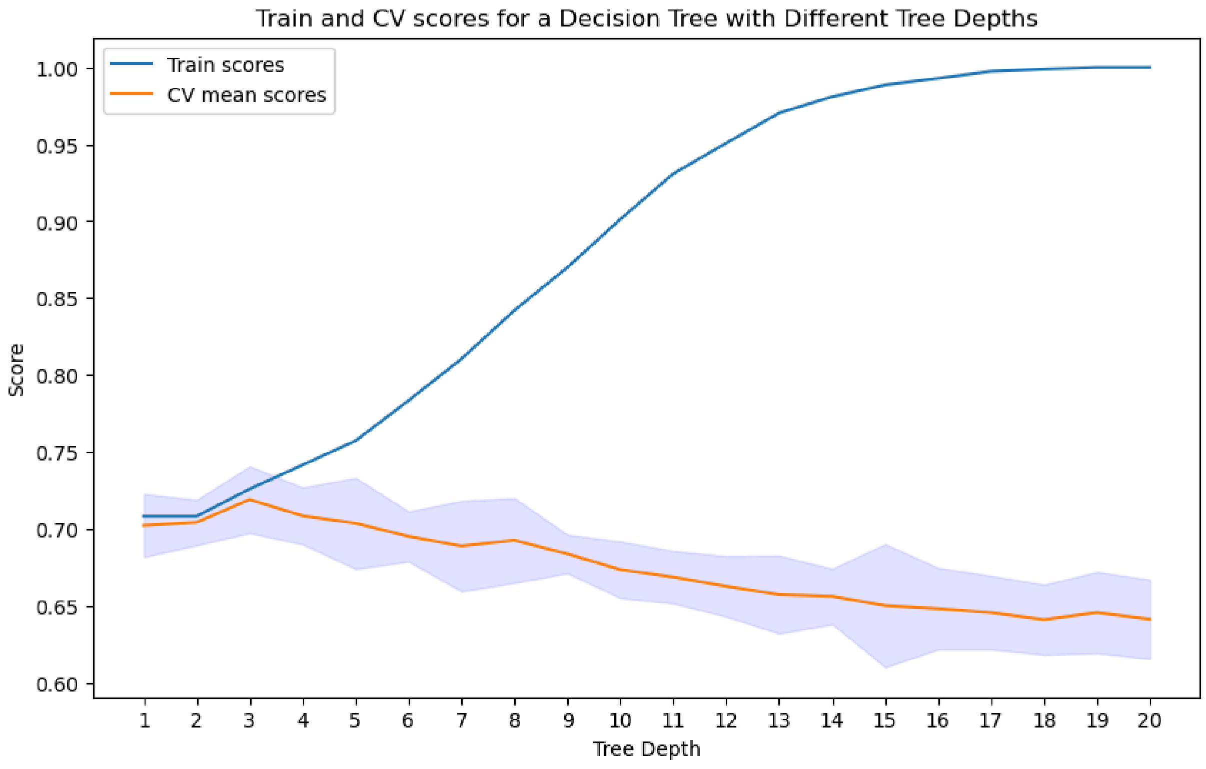
In [108…

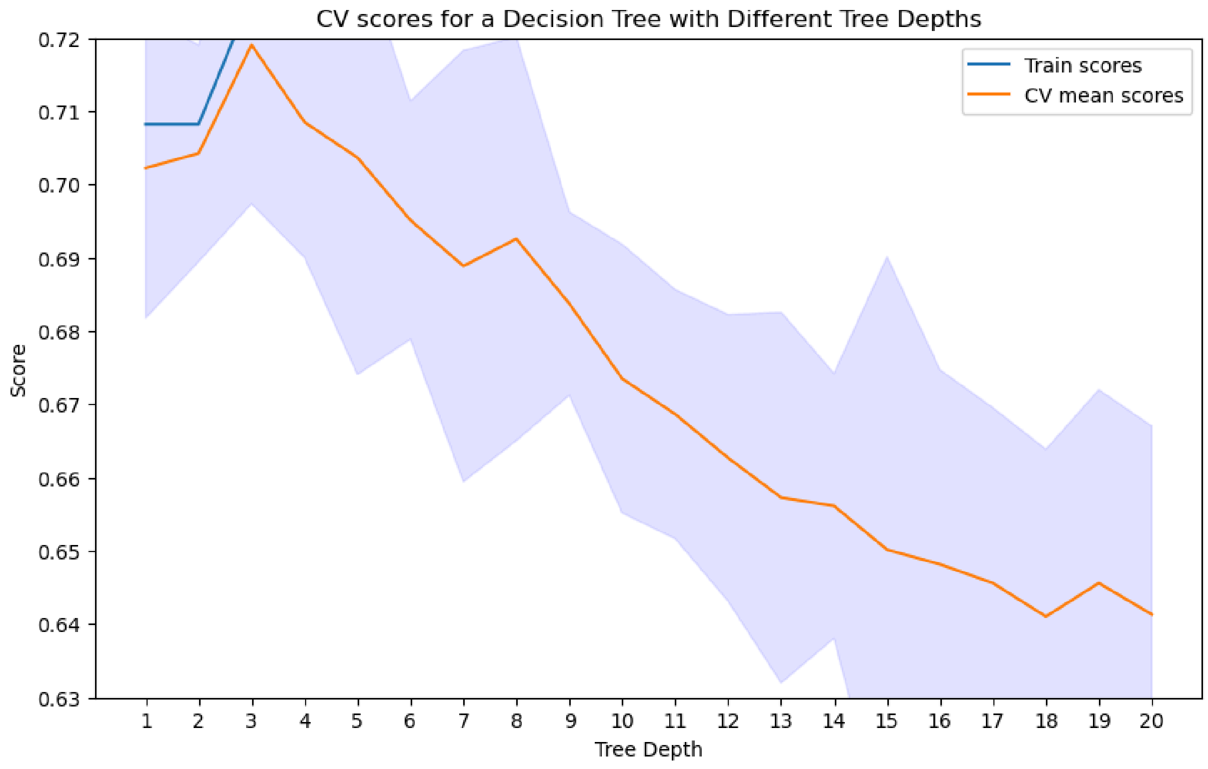
```
train_scores = []
cvmeans = []
cvstds = []

# Fit a decision tree to the entire training set for each depth from 1 to 20
for depth in range(1, 21):
    dt = DecisionTreeClassifier(max_depth=depth, random_state=0)
    dt.fit(X_train, y_train)

    # Evaluate on the entire training set
    train_scores.append(dt.score(X_train, y_train))

    # Perform 5-fold cross-validation with the entire training set
    cv_scores = cross_val_score(dt, X_train, y_train, cv=5)
    cvmeans.append(cv_scores.mean())
    cvstds.append(cv_scores.std())
```

In [109…

```
# Create a range for the depths
depths = range(1, 21)

# Plot 1: All scores
plt.figure(figsize=(10, 6))
plt.plot(depths, train_scores, label='Train scores')
plt.plot(depths, cvmeans, label='CV mean scores')
plt.fill_between(depths, np.array(cvmeans) - 2*np.array(cvstds), np.array(cv
plt.xticks(np.arange(1, 21, step=1))  # Set x-ticks from 1 to 20
plt.legend()
plt.xlabel('Tree Depth')
plt.ylabel('Score')
plt.title('Train and CV scores for a Decision Tree with Different Tree Depth
plt.show()
```

```
# Plot 2: Focus on validation performance
plt.figure(figsize=(10, 6))
plt.plot(depths, train_scores, label='Train scores')
plt.plot(depths, cvmeans, label='CV mean scores')
plt.fill_between(depths, np.array(cvmeans) - 2*np.array(cvstds), np.array(cv
plt.ylim(0.63, 0.72)
plt.xticks(np.arange(1, 21, step=1))  # Set x-ticks from 1 to 20
plt.legend()
plt.xlabel('Tree Depth')
plt.ylabel('Score')
plt.title('CV scores for a Decision Tree with Different Tree Depths')
plt.show()
```





In [110…
```
best_cv_depth = np.argmax(cvmeans) + 1  # Add 1 because depths start from 1

# Justification: The best depth is the one that maximizes the mean cross-val

# Fit a new decision tree on the entire training data using the best depth
best_cv_tree = DecisionTreeClassifier(max_depth=best_cv_depth, random_state=
best_cv_tree.fit(X_train, y_train)

# Store the train and test accuracies
best_cv_tree_train_score = best_cv_tree.score(X_train, y_train)
best_cv_tree_test_score = best_cv_tree.score(X_test, y_test)

# Print the best depth and accuracies
print(f"Best depth: {best_cv_depth}")
print(f"Train accuracy: {best_cv_tree_train_score}")
print(f"Test accuracy: {best_cv_tree_test_score}")
```

```
Best depth: 3
Train accuracy: 0.7258753202391118
Test accuracy: 0.7076222980659841
```

We see that a tree depth of 3 yields the best validation accuracy, which makes sense because as the depth increases, the tree begins to overfit.

Next, we use bagging on the most overfit depth (18) to help correct the overfitting of the decision tree and reduce the variance of the model.

In [111…
```
# Create a BaggingClassifier with a deep DecisionTreeClassifier
bagging_clf = BaggingClassifier(DecisionTreeClassifier(max_depth=18, random_
bagging_clf.fit(X_train, y_train)
```

```
# Store the train and test accuracies
bagging_train_score = bagging_clf.score(X_train, y_train)
bagging_test_score = bagging_clf.score(X_test, y_test)

# Print the accuracies
print(f"Train accuracy: {bagging_train_score}")
print(f"Test accuracy: {bagging_test_score}")
```

```
Train accuracy: 0.9988613720466838
Test accuracy: 0.726962457337884
```

We also fit a random forest classifier on the data, as another way of reducing overfitting but this time introducing randomness into the model creation process.

In [112…

```
# Create a RandomForestClassifier with different number of trees
best_train_score, best_test_score = 0, 0
best_n_trees = 0
train_scores = []
cv_scores_list = []
cv_std_list = []
for n_trees in [50, 100, 150, 200, 250, 300]:
    random_forest_clf = RandomForestClassifier(n_estimators=n_trees, random_
    random_forest_clf.fit(X_train, y_train)

    # Perform cross-validation
    cv_scores = cross_val_score(random_forest_clf, X_train, y_train, cv=5)
    cv_mean = cv_scores.mean()
    cv_std = cv_scores.std()

    # Store the train score, cv score and cv std
    train_scores.append(random_forest_clf.score(X_train, y_train))
    cv_scores_list.append(cv_mean)
    cv_std_list.append(cv_std)

    # Check if the current cross-validation score is the best
    if cv_mean > best_test_score:
        best_test_score = cv_mean
        best_n_trees = n_trees
```

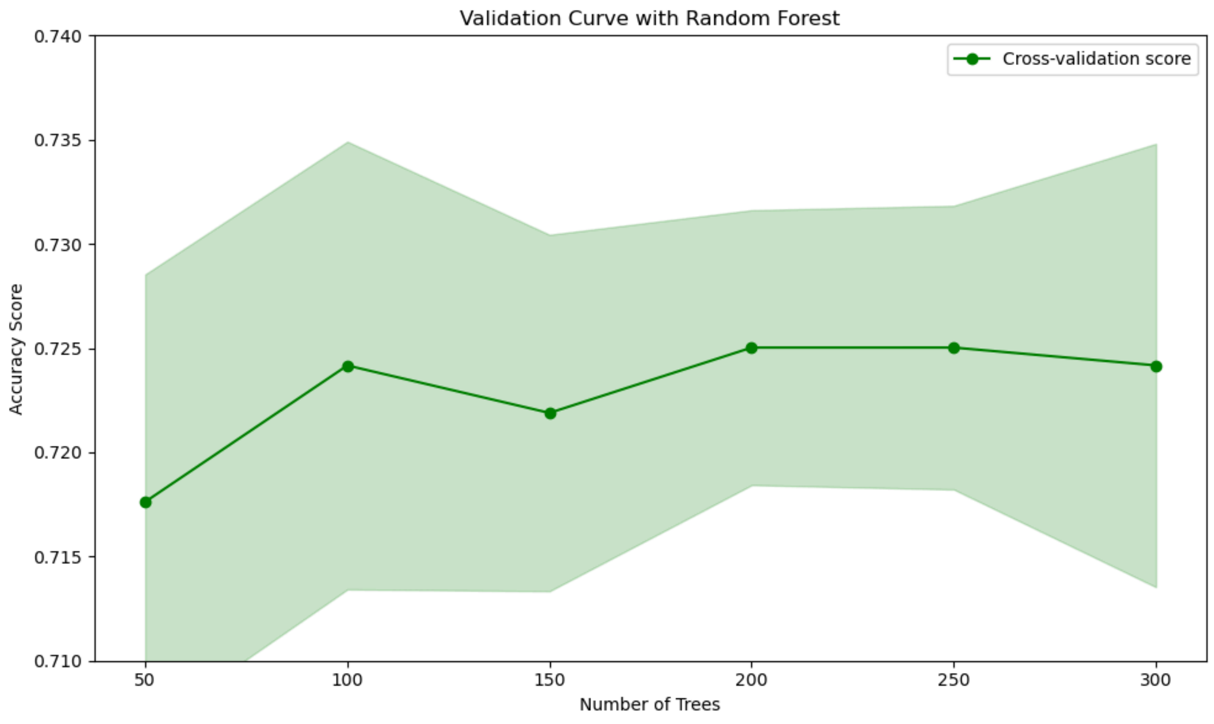
In [113…

```
# Fit the model with the best number of trees
random_forest_clf = RandomForestClassifier(n_estimators=best_n_trees, random
random_forest_clf.fit(X_train, y_train)

# Store the train and test accuracies
random_forest_train_score = random_forest_clf.score(X_train, y_train)
random_forest_test_score = random_forest_clf.score(X_test, y_test)

# Print the best number of trees and the corresponding test accuracy
print(f"Best number of trees: {best_n_trees}")
print(f"Best test accuracy: {random_forest_test_score}")
```

```
Best number of trees: 250
Best test accuracy: 0.7406143344709898
```

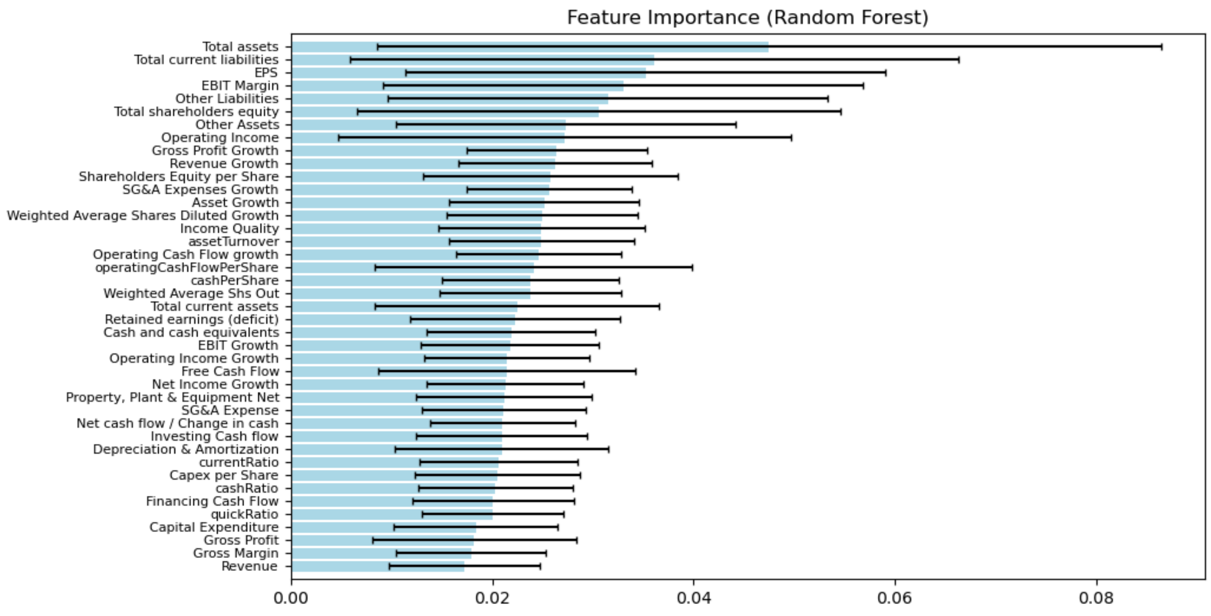
In [122…

```
# Plot the train and cross-validation scores
plt.figure(figsize=(10, 6))
plt.plot([50, 100, 150, 200, 250, 300], cv_scores_list, 'o-', color="g", lab
plt.fill_between([50, 100, 150, 200, 250, 300], np.array(cv_scores_list) - n
                 np.array(cv_scores_list) + np.array(cv_std_list), alpha=0.2

# Create plot
plt.title("Validation Curve with Random Forest")
plt.xlabel("Number of Trees")
plt.ylim(0.71, 0.74)
plt.ylabel("Accuracy Score")
plt.tight_layout()
plt.legend(loc="best")
plt.show()
```



In [114…

```
importances = random_forest_clf.feature_importances_
std = np.std([tree.feature_importances_ for tree in random_forest_clf.estima
indices = np.argsort(importances)[::-1]

# Get the feature names
feature_names = X.columns

# Plot the feature importances of the forest
plt.figure(figsize=(10, 6))
plt.title("Feature Importance (Random Forest)")
plt.barh(range(X_train.shape[1]), importances[indices][::-1],  # Reverse the
       color="lightblue", xerr=std[indices][::-1], align="center", capsize=2
plt.yticks(range(X_train.shape[1]), feature_names[indices][::-1], fontsize=8
plt.ylim([-1, X_train.shape[1]])
plt.show()
```



Finally, we also fit a gradient boosting classifier on the data, as an iterative model, to see if we can reduce variance in a different way.

In [115…

```python
# Create a GradientBoostingClassifier
boosting_clf = GradientBoostingClassifier(n_estimators=100, random_state=0)

# Define a grid of hyperparameters to search
param_grid = {
    'learning_rate': [0.01, 0.1, 0.2, 0.5, 1.0]
}

# Use GridSearchCV to find the best learning rate
grid_search = GridSearchCV(boosting_clf, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Get the best parameters
best_params = grid_search.best_params_
print(f"Best parameters: {best_params}")

# Fit the model with the best parameters
boosting_clf = GradientBoostingClassifier(n_estimators=100, random_state=0,
boosting_clf.fit(X_train, y_train)

# Store the train and test accuracies
boosting_train_score = boosting_clf.score(X_train, y_train)
boosting_test_score = boosting_clf.score(X_test, y_test)

# Print the accuracies
print(f"Train accuracy: {boosting_train_score}")
print(f"Test accuracy: {boosting_test_score}")
```

```
Best parameters: {'learning_rate': 0.1}
Train accuracy: 0.8334756618274979
Test accuracy: 0.7337883959044369
```

In [116…
```
# Create a DataFrame to store the accuracies
accuracies = pd.DataFrame({
    'Model': ['Best CV Depth DecisionTreeClassifier', 'BaggingClassifier', '
    'Train Accuracy': [best_cv_tree_train_score, bagging_train_score, random
    'Test Accuracy': [best_cv_tree_test_score, bagging_test_score, random_fo
})

accuracies
```

Out[116…

	Model	Train Accuracy	Test Accuracy
0	Best CV Depth DecisionTreeClassifier	0.725875	0.707622
1	BaggingClassifier	0.998861	0.726962
2	RandomForestClassifier	1.000000	0.740614
3	GradientBoostingClassifier	0.833476	0.733788

The RandomForestClassifier model performed the best with a train accuracy of 0.9997 and a test accuracy of 0.7338. This might be due to the fact that RandomForestClassifier is an ensemble learning method that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes of the individual trees. This makes it a very powerful model capable of handling a large dataset with high dimensionality.

The GradientBoostingClassifier also performed very well with a train accuracy of 0.833476 and a test accuracy of 0.733788. This model builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions -- thus, it is also a very powerful model when dealing with large amounts of data and high dimensionality.

The BaggingClassifier had a very high train accuracy of 0.9974 but a slightly lower test accuracy of 0.7235. This indicates that the model may have overfit the training data. BaggingClassifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions to form a final prediction. This can lead to high variance if the base classifier is not robust enough.

The Best CV Depth DecisionTreeClassifier had the lowest train and test accuracies of 0.7259 and 0.7076 respectively. Decision trees are simple to understand and interpret, but they can easily overfit the data and have poor prediction performance if the depth of the tree is not properly tuned.

Also, we took a look at the Random Forest model's ranking of feature importance, shown in the bar graph above. Total assets was top, which is interesting because total assets was eliminated during lasso regularization in the logistic regression model. This result may not be completely accurate, though, since the standard error bar for this feature is so wide --- the results may be skewed by outliers or just have high variance. It does make sense for total assets to be a good predictors of stock price increase, but total assets can vary so much across industry that it doesn't seem very accurate. Total current liabilities being a strong predictor is also counterintuitive but actually makes some sense, as high total current liabilities may indicate efficient use of capital (financing with debt over equity), which has many tax benefits. EPS, the next highest predictor, was the top predictor in the logistic regression with lasso, which strengthens its argument as a strong predictor.

Part II: Advanced Models

Return to contents

Neural Networks

Return to contents

Finally, we trained neural networks on all of the original features (after accounting for missingness and outliers). Since the models took much longer to train, we added the code in separate files to the assignment.

We trained the following models:

- model_bagging.py: neural network CV w/ bagging
- model.py: neural network CV (with optional small parameters for shorter training time)
- model_parallel.py: neural network CV (with parallel computing to speed up the process)
- model_RF.py: neural network random forests CV
- svm.py: SVM CV

General Model

Our general model.py file includes the base neural network we tried to implement. The file includes a test implementation of GridSearchCV with a neural network (with parameters hidden sizes and droupout rate). The next section of code includes a much more extensive test, with these parameters tested:

In [117…
```
'''
param_grid = {
    'lr': [0.01, 0.1, 0.001],
    'max_epochs': [10, 20, 30],
    'module__dropout_rate': [0.3, 0.5, 0.7],
    'module__hidden_sizes': [
        [64, 32],
        [128, 64, 32],
        [32, 32, 32],
        [256, 128, 64, 32],
        [64, 64, 64],
    ],
    'optimizer': [optim.Adam, optim.SGD, optim.RMSprop],
    'batch_size': [16, 32, 64],
    'module__num_layers': [2, 3, 4],
    'optimizer__weight_decay': [0.01, 0.001, 0.0001],
    'optimizer__momentum': [0.9, 0.95, 0.99],
    'module__activation_func': [F.relu, F.leaky_relu, F.elu, F.sigmoid],
}'''
```

Out[117…

```
"\nparam_grid = {\n    'lr': [0.01, 0.1, 0.001],\n    'max_epochs': [10, 2
0, 30],\n    'module__dropout_rate': [0.3, 0.5, 0.7],\n    'module__hidden_
sizes': [\n        [64, 32],\n        [128, 64, 32],\n        [32, 32, 3
2],\n        [256, 128, 64, 32], \n        [64, 64, 64],       \n    ],\n
'optimizer': [optim.Adam, optim.SGD, optim.RMSprop],\n    'batch_size': [1
6, 32, 64],\n    'module__num_layers': [2, 3, 4],\n    'optimizer__weight_d
ecay': [0.01, 0.001, 0.0001],\n    'optimizer__momentum': [0.9, 0.95, 0.9
9],\n    'module__activation_func': [F.relu, F.leaky_relu, F.elu, F.sigmoi
d],\n}"
```

With all these parameters, there are over 130,000 models that can be created. The best five-fold CV model had the following parameters, with a test accuracy of 0.7235, outperformed only by the random forest.

- Batch Size: 32,
- Learning Rate: 0.01,
- Max Epochs: 20,
- Dropout Rate: 0.3,
- Hidden Sizes: [64, 32],
- Number of Layers: 3,
- Optimizer: <class 'torch.optim.sgd.SGD'>

We also modified the code in model_parallel.py to take advantage of parallel processing to increase the training speed.

Neural Networks with Bagging

Next, we decided to try bagging in Neural Networks. We used the same approach used in class, but with Neural Networks. The code allows the user to enter a list of numbers of models, and then for each number, it runs a bagging model and returns the test training.

After running this on 5, 10, 20, 30, 40, and 50 models, here was our result:

Number of Models	Test Accuracy
5	0.6997
10	0.7144
20	0.7133
30	0.7076
40	0.7110
50	0.7110
60	0.7144

Neural Networks with Random Forests

Next, we implemented random forests in Neural Networks as well. We built onto the approach used in class. The code interface works similar to that above, with the user being able to enter their own list of numbers.

We trained RFs with 5, 10, 15, and 20 neural networks. Here are our results:

Number of Models	Test Accuracy
5	0.7042
10	0.7224
15	0.7144
20	0.7144

Note on Bagging and Random Forests

Note that we no longer used CV Grid Search here. In the future, it might be better if we introduce CV validation Grid Search in each training step. This would likely increase the performance of the model!

Reflection

The results came to around 0.70-0.72 accuracy for each of these models, with the standard neural network CV with 20 epochs performing the best at an accuracy of 0.72.

Neural networks might be a good fit for this problem due to their ability to model complex, non-linear relationships, which are common in financial data. They can capture interactions between different financial indicators that simpler models might miss. Additionally, neural networks can learn to recognize patterns in the data, which can be particularly useful when predicting financial trends. However, they can be computationally intensive and may require a large amount of data to train effectively.

Despite their potential, the neural networks might not be performing as well in this case due to several reasons. First, the complexity of these models can lead to overfitting, especially if the network architecture is not properly designed. Second, the training process of neural networks is computationally intensive and was very slow. This can be a problem if the dataset is large or if the model needs to be retrained frequently. Lastly, neural networks require a large amount of data to train effectively. If the dataset is not large enough, the models may not be able to learn the underlying patterns in the data, leading to poor performance.

Next, we will continue to explore these models and fine tune them to better performance.

Support Vector Machine

We also took a crack at an SVM model! The impelmentation was very similar to our base Neural Network CV Grid Search; we tested for these parameters:

In [118…

```
'''
param_grid = {
    'C': [0.1, 1, 10],
    'gamma': [0.01, 0.1, 1],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'degree': [2, 3, 4],  # Only for poly kernel
    'coef0': [0.0, 1.0, 2.0],  # Only for poly and sigmoid kernels
    'shrinking': [True, False],
    'tol': [1e-3, 1e-4, 1e-5],
    'class_weight': [None, 'balanced'],
    'decision_function_shape': ['ovo', 'ovr'],
}
'''
```

Out[118…

```
"\nparam_grid = {\n    'C': [0.1, 1, 10],\n    'gamma': [0.01, 0.1, 1],\n
'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],\n    'degree': [2, 3, 4],
# Only for poly kernel\n    'coef0': [0.0, 1.0, 2.0],  # Only for poly and
sigmoid kernels\n    'shrinking': [True, False],\n    'tol': [1e-3, 1e-4, 1
e-5],\n    'class_weight': [None, 'balanced'],\n    'decision_function_shap
e': ['ovo', 'ovr'],\n}\n"
```

This was over 33,000 models! We found that the best model had a test accuracy of 0.7076. The best parameters were:

- C: 1
- Class Weight: None
- Coefficient 0: 0.0

- Decision Function Shape: ovo
- Degree: 2
- Gamma: 0.01
- Kernel: rbf
- Shhrinking: True
- Tol: 0.001

In the next steps, we will finetune better and hopefully find better models!