Financial Data Science (FIN42110)

Group Project Title:

Financial Data Science for Implementing Pairs Trading Strategy

---

Presented by: Group 6

Kuntoji, Rohan - 22202093 (MSc. Financial Data Science)

Parary, Aman - 18417714 (MSc. Financial Data Science)

Fernandes, William - 22201553 (MSc. Financial Data Science)

Introduction

According to (Göncü and Akyıldırım, 2016), financial markets heavily rely on algorithmic trading due to the large number of assets and fast information flow. This trend requires investors to use advanced quantitative methods for generating trading signals. Statistical arbitrage is a type of trading strategy that can generate guaranteed profits through quantitative models. Pairs trading is a popular statistical arbitrage technique that capitalizes on temporary market patterns between stocks with similar past performance but opposite market trends. Traders identify stock pairs and exploit short-term relative mispricing by buying the underperforming stock and taking a short position in the overperforming stock. This approach establishes both long and short positions in two stocks with the same dollar amount, potentially resulting in a market-neutral position that mitigates risks (Lu et al., 2021).

In this project we used financial and textual data of S&P 500 constituents to conduct financial data analysis to generate clusters of stocks. These clusters are created because relying on conventional sectors in the S&P 500 might not be advantageous since these sectors are founded on obsolete and random categorizations of industries. The classification of companies into sectors and sub-sectors is based on the Global Industry Classification Standard (GICS), which may have its limitations as it may not precisely portray the business models or growth prospects of particular companies. So we will use the clusters generated through our analysis and arrive at possible pairs of stocks for portfolio creation and trading.

Section 1: Novel Data Set Collection

For the Novel Data Set collection, different datasets on S&P 500 constituents were collected from various sources and pooled together to produce a novel database.

- Basic profile data for the S&P 500 constituents were collected from a public wiki page. A total of 503 stocks were identified, forming our stock universe. This data includes information such as company ticker, name, sector, sub-industry, headquarters, and business descriptions. The data is stored in the `pairs_trading.db` database under the table `stocks_profile`.
- Historical monthly price data from Jan 2010 to March 2023 for the defined stock universe was fetched and stored in `pairs_trading.db` database. The table name is `stock_hist_price`, consisting of $1,590,412$ rows and 12 columns.
- Later, annual historical data for 54 Financial Ratios data of all stocks in the defined universe of S&P 500 constituent stocks was fetched from 2010 to 2023 in `pairs_trading.db` database. The table name is `stock_hist_ratios`. There were 502 stocks in the historical ratios data, resulting in 354,468 (54*502*13) data points for the financial ratios
- The common tickers from the above three datasets were then fetched, resulting in a common universe of 502 stock tickers.

Section 2: Database Creation and Querying - Summary Statistics

- We created three database tables: `stocks_hist_price`, `stock_hist_ratios`, and `stocks_profile` to store collected data from the novel dataset.
- Executed queries to extract data for each table to perform exploratory data analysis, data cleaning, and model building.
- Additionally, we generated summary statistics using queries to gain deeper insights into our novel data set.
- For example, 73 companies from both Industrials and Financials sectors are among the S&P 500 constituents, as shown in Table 3.
- The majority of S&P 500 companies are based in the USA, as shown in Table 2.

- The highest total trading volume across all tickers occurred on 2014-06-02, as shown in Table 1.

Date	Total_Volume
2014-06-02	997828800
2022-09-30	996797482
2011-08-24	993358336
2015-08-24	991392689
2011-08-04	988146567
2020-09-04	987072445
2023-02-02	986906699
2010-07-16	986740951
2015-12-29	986416387
2021-12-28	984775851

Table 1: Total Trading Volume by Date

Location	Num_Companies
California	69
New York	53
Texas	45
Illinois	33
Massachusetts	22
Pennsylvania	19
Ohio	19
Georgia	18
North Carolina	16
Florida	16

Table 2: Number of Companies by Location

Sector	Num_Companies
Industrials	73
Financials	73
Information Technology	66
Health Care	65
Consumer Discretionary	53

Table 3: Number of Companies by Sector

Section 3: Data Cleaning, Checking, and Organization

- A reduced number of target financial ratios were selected for the analysis.
- Data cleaning was conducted on the raw historical financial ratios data. A KNN imputer was used to impute missing values in the dataset, reducing the missing values to zero.
- To reduce the dimensionality of the data, PCA was employed. The Scree plot in Figure 1, suggested an optimal number of 12 components. Subsequently, a new PCA was fit based on these 12 optimal components.
- A few eigenportfolios are utilised to reflect the systematic risk of the entire stock universe using the limited collection of series provided by PCA to mimic a much larger one (Krauss, 2016).
- The textual data was cleaned and organised through stop word removal, stemming, lemmatisation, and tokenisation.
- In Section 1, we collected basic profile data for the S&P 500 constituents, which included business descriptions. The business description text was cleaned and wrangled to extract 2,927 unique words from a corpus of business descriptions across all stocks. Tickers with empty business descriptions were removed from the dataset.

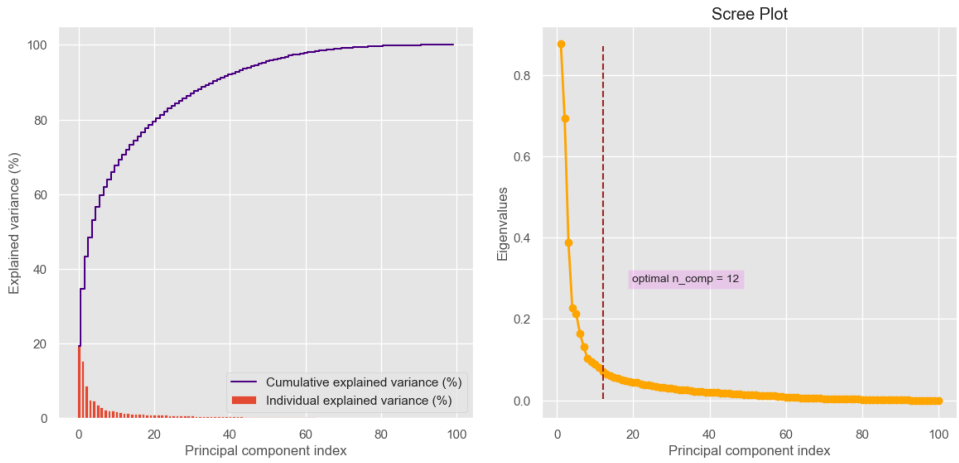

Figure 1: Scree plot indicating 12 optimal principal components, chosen to reduce data dimensionality

Section 4: Data Visualisation

- In Figure 2, which illustrates a feature space correlation map (consisting of 12-PCA components and 80 topics derived from LSA), all features in the space exhibit close to zero correlation. This suggests that the feature space is well-suited for model fitting. Additionally, the attributes have been standardized, now possessing a mean of 0 and a standard deviation of 1.
- The data has no missing values now nor duplicate entries. Outliers have been identified but not treated, and they are assumed to be actual data points & not anomalies in this context for modelling.

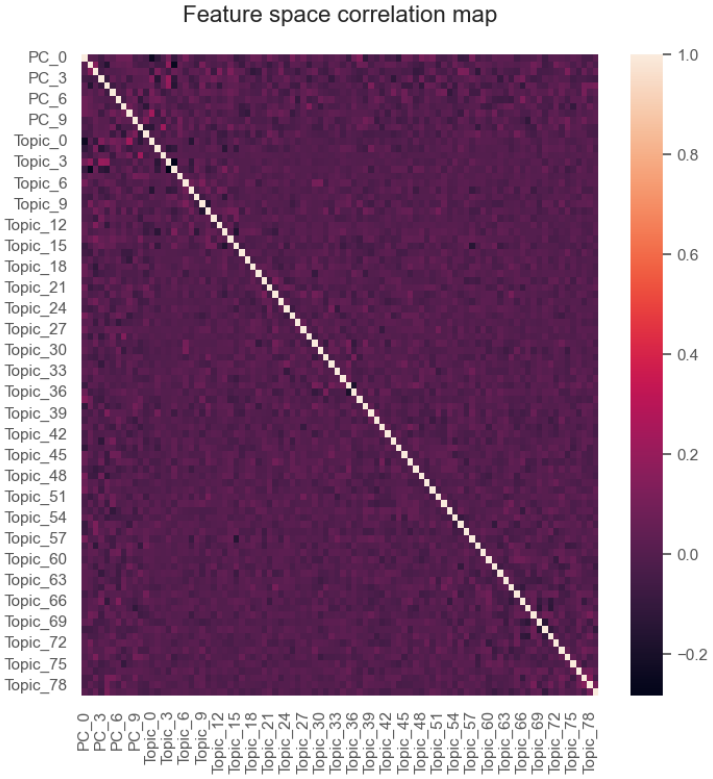

Figure 2: Feature space correlation map with 12-PCA & 80 LSA topics, showing near-zero correlation, ideal for model fitting. Attributes standardized (mean=0, SD=1).

- We used boxplots in Figure 3 to examine the organization of the final resulting clusters based on the defined target financial ratios. The boxplots provided insights into the median, outliers, and skewness of the clusters. For example, we observed that cluster -1 had many outliers across almost all the target financial ratios, indicating an unclustered class. Additionally, several clusters exhibited skewness in various ratios. For example, the boxplots of asset turnover for cluster 2 showed positive skewness, suggesting potential differences in the profitability of companies in this cluster.

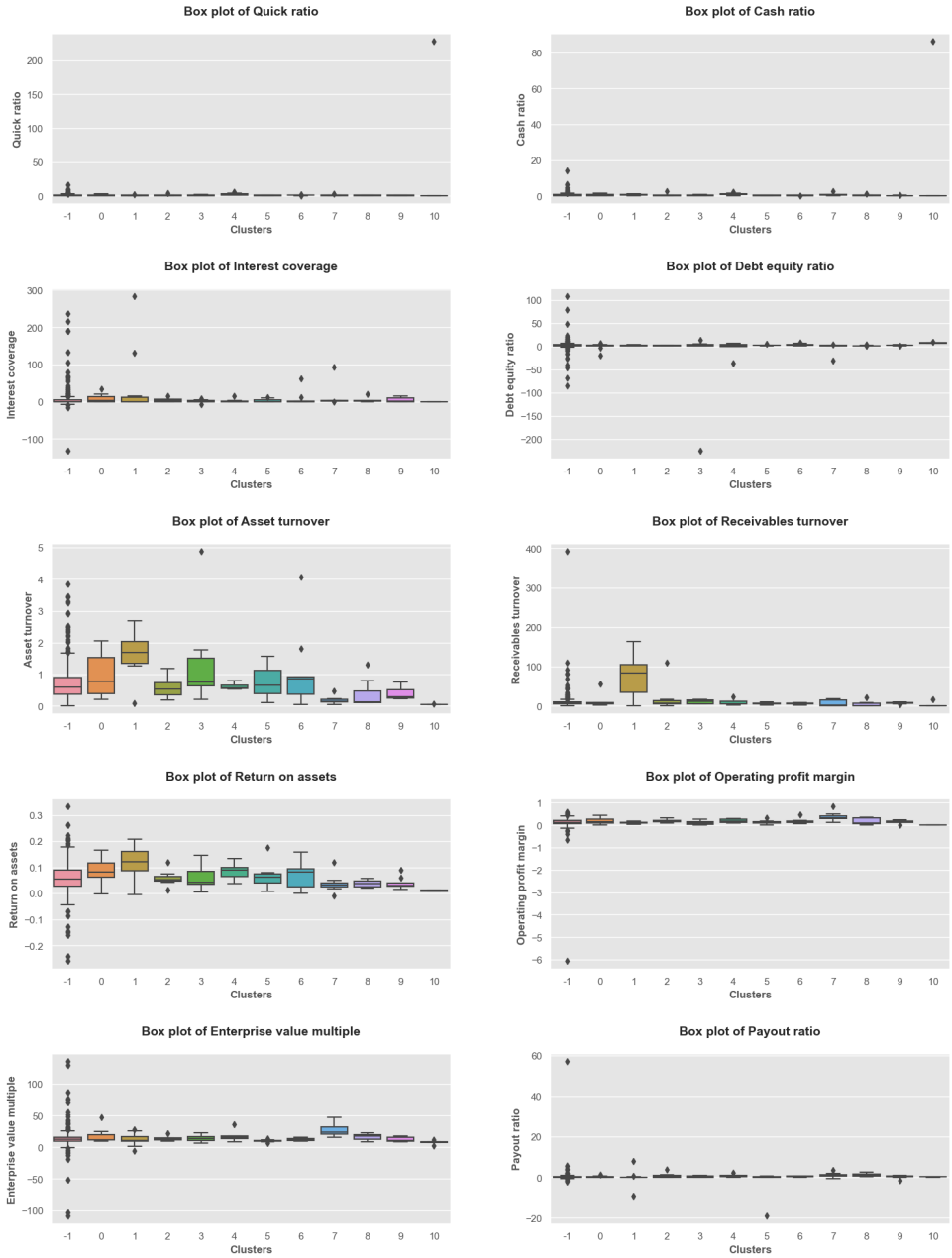

Figure 3: Boxplots examined cluster organization based on financial ratios, revealing median, outliers, and skewness. Cluster -1 had numerous outliers, indicating an unclustered class. Skewness was observed in various ratios across clusters, e.g., positive skewness in asset turnover for cluster 2, suggesting profitability differences.

- A word cloud plot, as shown in Figure 4 provides an insight into business descriptions of companies classified in each of the 10 final clusters.

Figure 4: Word cloud plot reveals business descriptions for companies in each of the 10 final clusters.

Section 5: Textual Analysis

- The cleaned textual data were vectorised using the TF-IDF method to get a sparse DTM on the cleaned data.
- In Figure 5, the feature matrix from textual data appears primarily empty, even when sampling a random feature set of size 10. This data sparsity may cause issues during model training, as low-weighted features could skew cluster formations based on dataset ratios. We must reduce the feature space to prevent this while preserving explained variance among features.

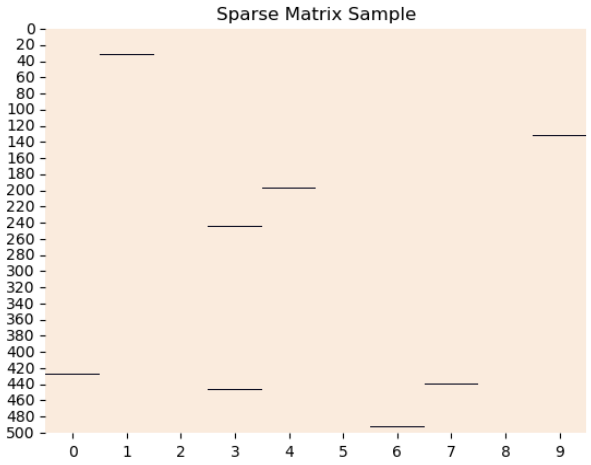

Figure 5: Textual data's sparse feature matrix may impact model training; reducing feature space while preserving explained variance is crucial.

- Latent Semantic Analysis (LSA) was applied to reduce the dimensionality of the TF-IDF sparse matrix and get a dense matrix. This application reduced the dimensionality from 10000 to 80 features, and the total variance explained by the reduced features was about 24%.
- Figure 6 shows plots of (ngrams) within each topic.

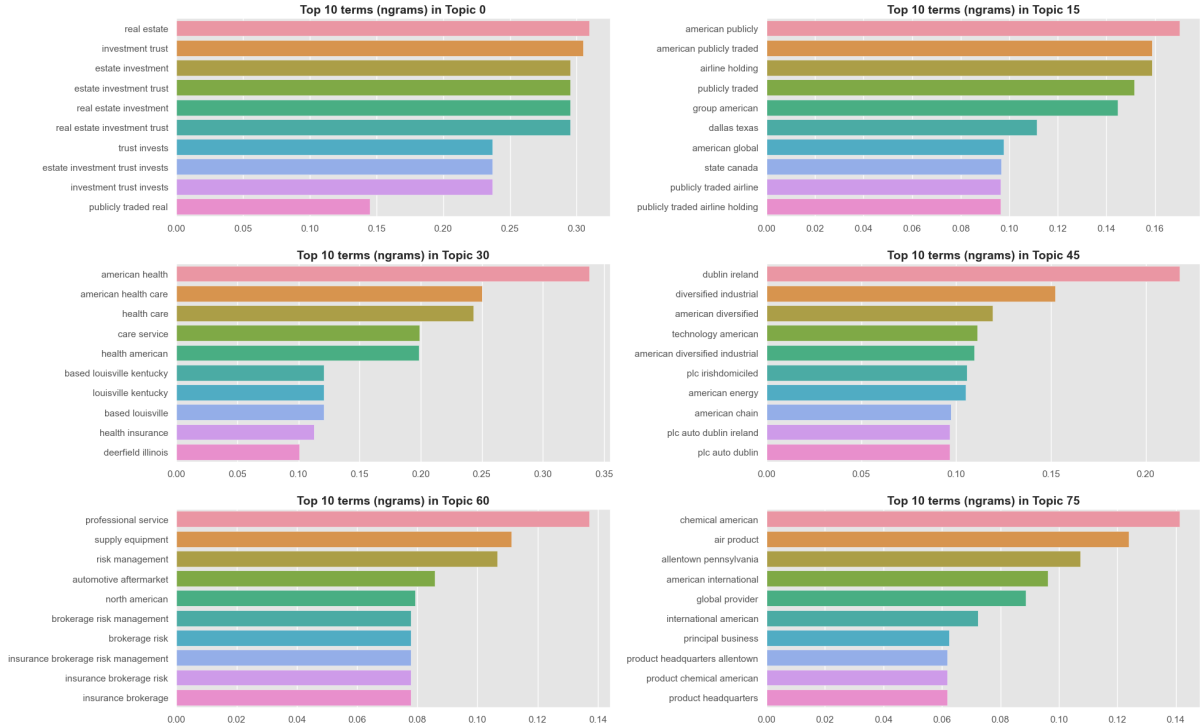

Figure 6: Displays bar charts of the top 10 n-grams within each topic.

Section 6: ML based predictive/explanatory modelling: Methodology, analysis and results

- We applied three clustering methods for predictive modeling, dividing data objects into non-overlapping groups with each object belonging to only one cluster.

    1. Partitional clustering: We used the k-means algorithm, a common unsupervised learning technique. The optimal K value was determined using the Elbow test and Silhouette score analysis (as shown in Figure 7), with $K = 17$ yielding better results. The t-distributed stochastic neighbor embedding (t-SNE) plot in Figure 8 is a statistical method for visualizing high-dimensional data by giving each data point a location in a two or three-dimensional map.

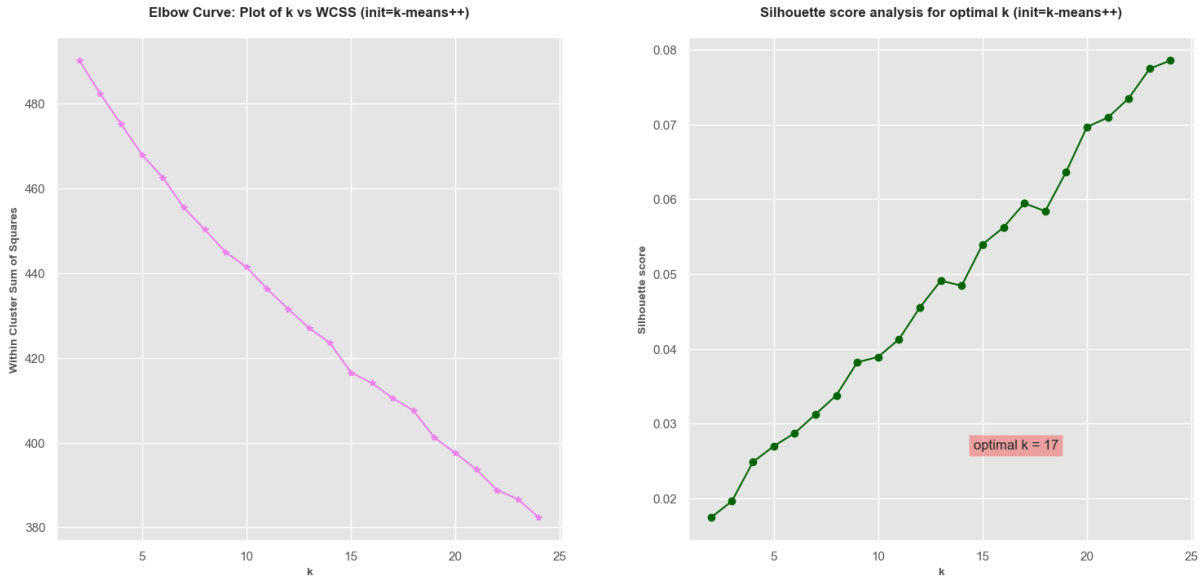

Figure 7: After conducting the Elbow test and Silhouette score analysis, it was found that K=17 provided the most favorable results, indicating it to be the optimal value for the given dataset

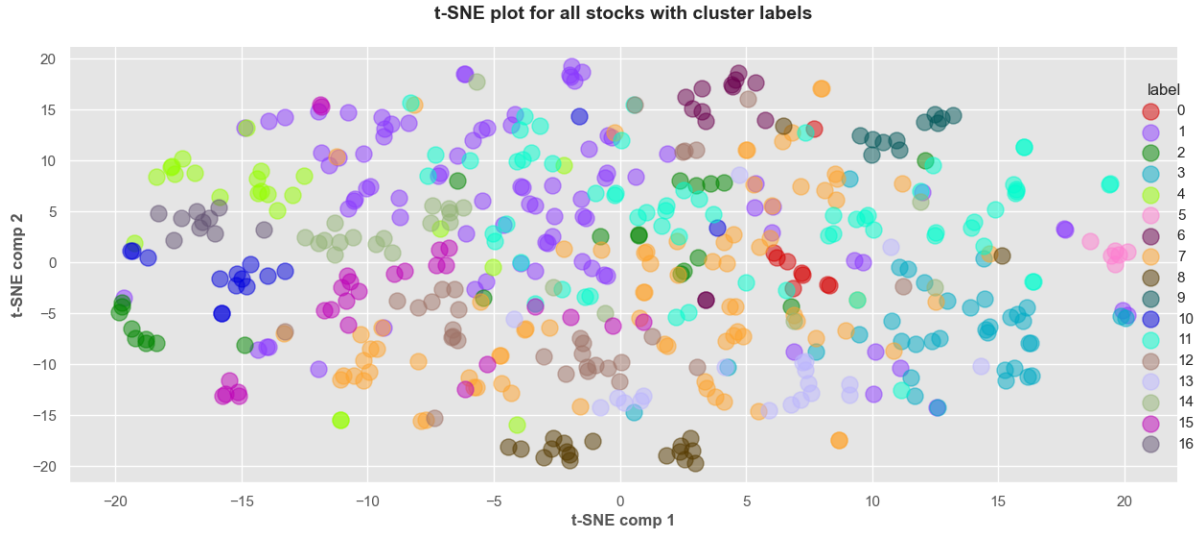

Figure 8: K-Means clustering with 17 clusters shows no clear formation of clusters, it suggests that the data may not have clear separation between the groups.

2. Density-based clustering (Optics Model): This method assigns clusters based on data point density in a region. Clusters are assigned where there are high densities of data points separated by low-density regions. In Figure 9, the t-SNE plot showed better clustering, distinct boundaries, and cohesive clusters, making this model preferable. The cluster count bar chart in Figure 10 shows the number of stocks in each cluster classified by the Optics model. The t-SNE plot is also justified by the outliers in the data as seen in the boxplots of financial ratios. The algorithm performed well to identify these outliers and not cluster them (as most outliers are seen in class label -1).

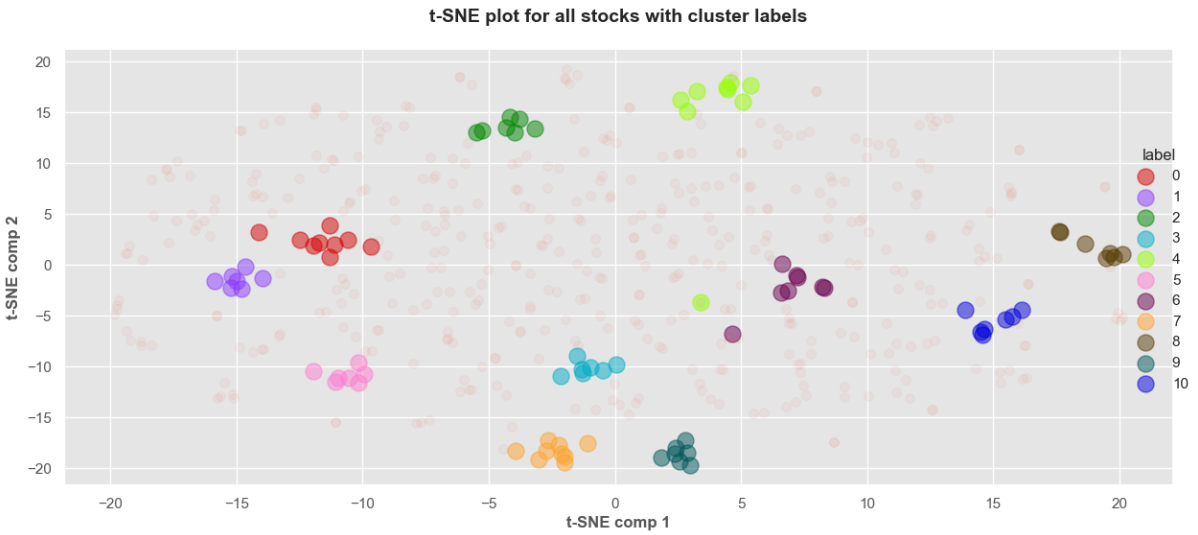

Figure 9: The t-SNE plot exhibits clear separation between clusters, well-defined boundaries, and compact clusters, indicating that this model performs favorably for the given dataset.

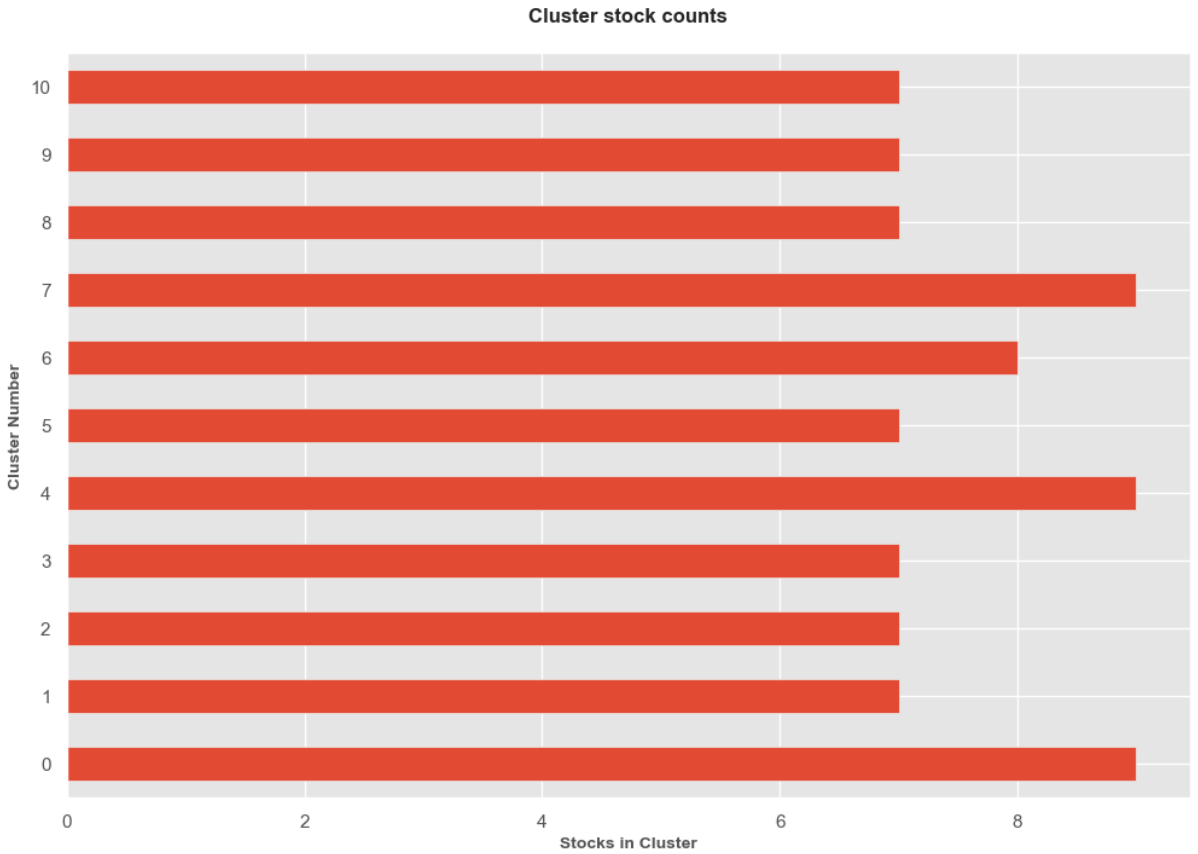

Figure 10: Bar chart displaying 11 clusters with stocks ranging between 7 and 9 per cluster.

Figure 11: Cluster stock price movements: sample plots of 4 least dense clusters

3. Hierarchical clustering (Agglomerative clustering): This bottom-up approach merges the two most similar points until all have been combined into a single cluster. In Figure 12, the t-SNE plot of Hierarchical clustering did not show improvement over the previous two methods. Hence, the Optics model was the preferred model from our analysis.

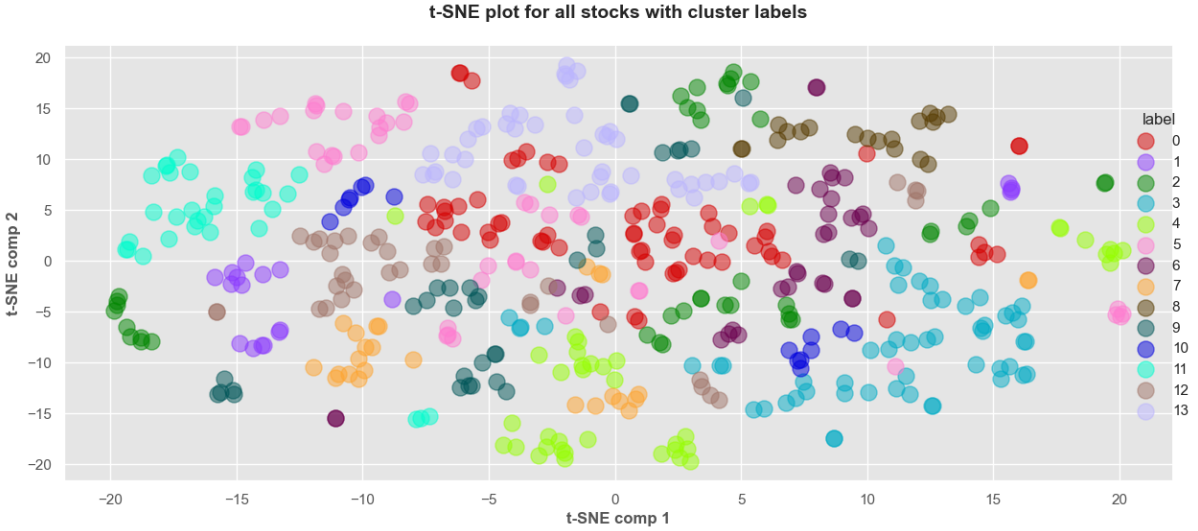

Figure 12: Plot does not show any significant improvement in cluster segmentation, indicating a lack of clear grouping within the dataset.

- Figure 13 displays the sector-wise breakdown of clusters obtained from the OPTICS model, which includes stocks from the GICS sector segmentation.
- Many Financial, Tech, and Health sector stocks had missing or outlier values, leading to their exclusion from clustering. For example, 60-70% of Finance stocks had missing values for key ratios like interest coverage ratio.

cluster	-1	0	1	2	3	4	5	6	7	8	9	10
sector												
Communication Services	23	0	0	0	1	0	0	0	0	0	0	0
Consumer Discretionary	47	1	2	0	2	0	0	0	0	0	0	0
Consumer Staples	32	0	1	3	0	0	0	1	0	0	0	0
Energy	21	0	1	0	0	0	0	1	0	0	0	0
Financials	62	1	0	0	0	1	1	0	0	0	0	7
Health Care	57	2	0	2	2	0	0	0	0	0	0	0
Industrials	59	1	2	1	1	4	1	1	0	1	0	0
Information Technology	61	3	1	0	1	0	0	0	0	0	0	0
Materials	21	0	0	1	0	2	4	0	0	1	0	0
Real Estate	9	0	0	0	0	0	0	0	9	5	7	0
Utilities	19	1	0	0	0	2	1	5	0	0	0	0

Figure 13: OPTICS clustering results for GICS sector segmentation. Some sectors (e.g. Finance, Tech, Health) were excluded due to missing or outlier values in key ratios, such as the interest coverage ratio.

- According to (Krauss, 2016) cointegration simulates long-term price interdependence while correlation depicts short-term linear reliance in returns. As a result, , the cointegration methodology offers a larger potential for detecting real long-term equilibrium linkages between various assets.
- We analyzed the clusters determined through the Optics model to arrive at the statistically cointegrated pairs. Cointegration analysis is a statistical method used to assess the long-term equilibrium relationship between non-stationary time series data. The Augmented Dickey-Fuller (ADF) test is commonly used to determine whether a sample time series is stationary or not by testing the presence of a unit root, or random walk, in the data.
- According to (Qazi, Rahman, and Gul 2015), implementing a successful pairs trading strategy requires selecting pairs of stocks that demonstrate a long-run equilibrium relationship and short-run relations that ensure mean reversion. Mean reversion is vital because if a divergence from the equilibrium position creates an arbitrage opportunity and a trade is opened, there must be a subsequent convergence to restore equilibrium and close the trade to earn arbitrage profits. Therefore, choosing the right pairs of stocks that exhibit these characteristics is essential for a successful pairs trading strategy.
- Using the ADF test and p-values, we arrived at 9 cointegrated pairs (shown in Figure 14) where the null hypothesis was rejected at a 0.05 level of significance (95% confidence level), indicating a meaningful and statistically significant relationship between the cointegrated pairs.

Figure 14: Displays 9 cointegrated pairs with a statistically significant relationship, as determined by the ADF test at a 95% confidence level.

Section 7: Business Impact Analysis

- In our pairs trading using Bollinger Bands, the strategy is based on trading the difference between the prices of two related instruments. The Bollinger Bands are used to determine the upper and lower price bands around the moving average of each instrument. When the price difference between the two instruments crosses above the upper Bollinger Band, it is considered an opportunity to sell the spread (short the overpriced instrument and long the underpriced instrument). Conversely, when the price difference crosses below the lower Bollinger Band, it is considered an opportunity to buy the spread (long the overpriced instrument and short the underpriced instrument). To implement this strategy, the code would involve calculating the Bollinger Bands for each instrument, calculating the difference in price (spread) between the two instruments, and using this difference to generate the entry signals for the trading strategy.
- The pairs trading strategy seems to have outperformed the benchmark (SPY - tracks S&P500) by a significant margin. When comparing portfolio performance to the benchmark, the Sharpe ratio provides insight into the risk-adjusted returns of each approach. In this case, our portfolio of two cointegrated pairs outperformed the benchmark with a Sharpe ratio of 0.1456 compared to the benchmark's Sharpe ratio of 0.0424. This suggests that the pairs trading strategy generated a higher return per unit of risk taken on, making it a more attractive investment strategy from a risk-adjusted return perspective. It also generated a cumulative return of 35% over the calendar year 2020 as opposed to the benchmark return of 20%.
- It is important to note, however, that the risk profile of the pairs trading strategy may differ from that of the benchmark, as it involves trading specific pairs of assets. Therefore, investors should consider their risk tolerance and investment objectives before adopting this strategy. Additionally, a longer time horizon and analysis of the specific pairs traded would be necessary to better assess the sustainability and potential effectiveness of the pairs trading (ensuring that cointegration of the pair holds good) strategy over the long term.

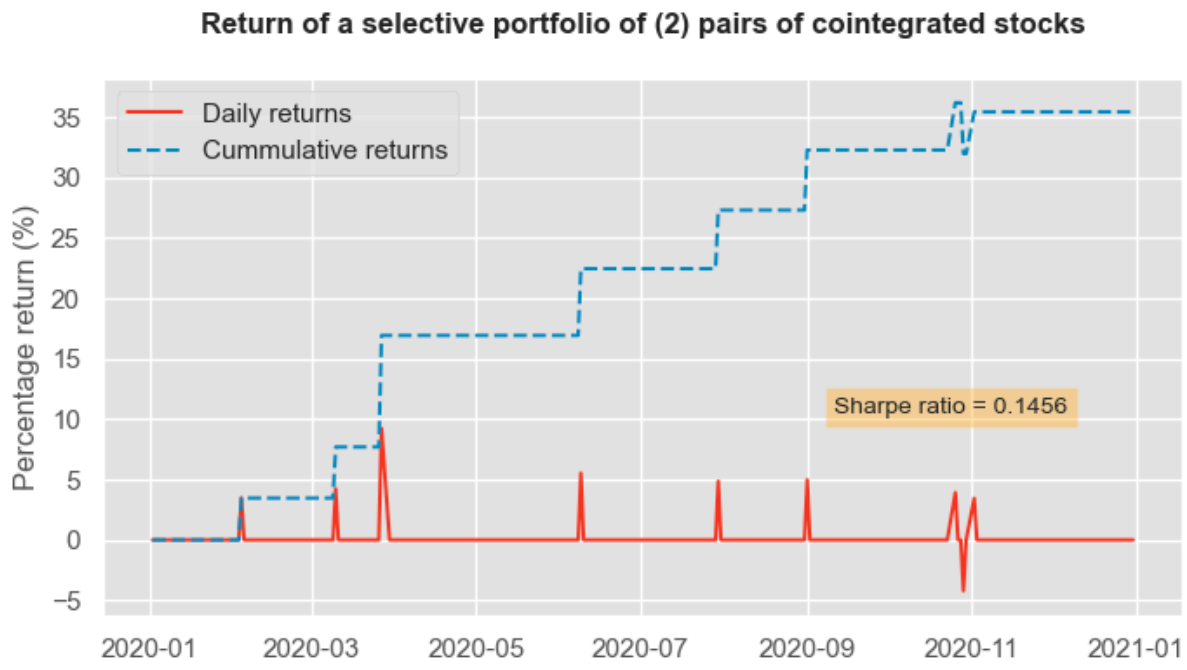

Figure 15: Portfolio performance over the CY 2020 (COVID-19 period)

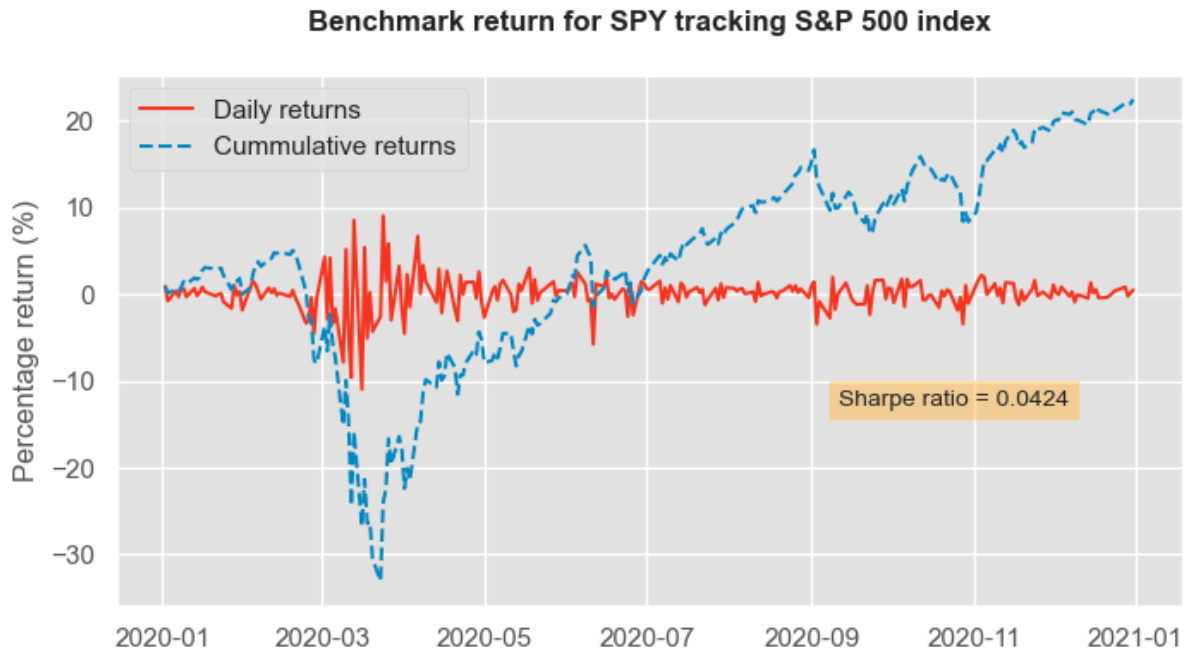

Figure 16: Benchmark performance over the CY 2020 (COVID-19 period)

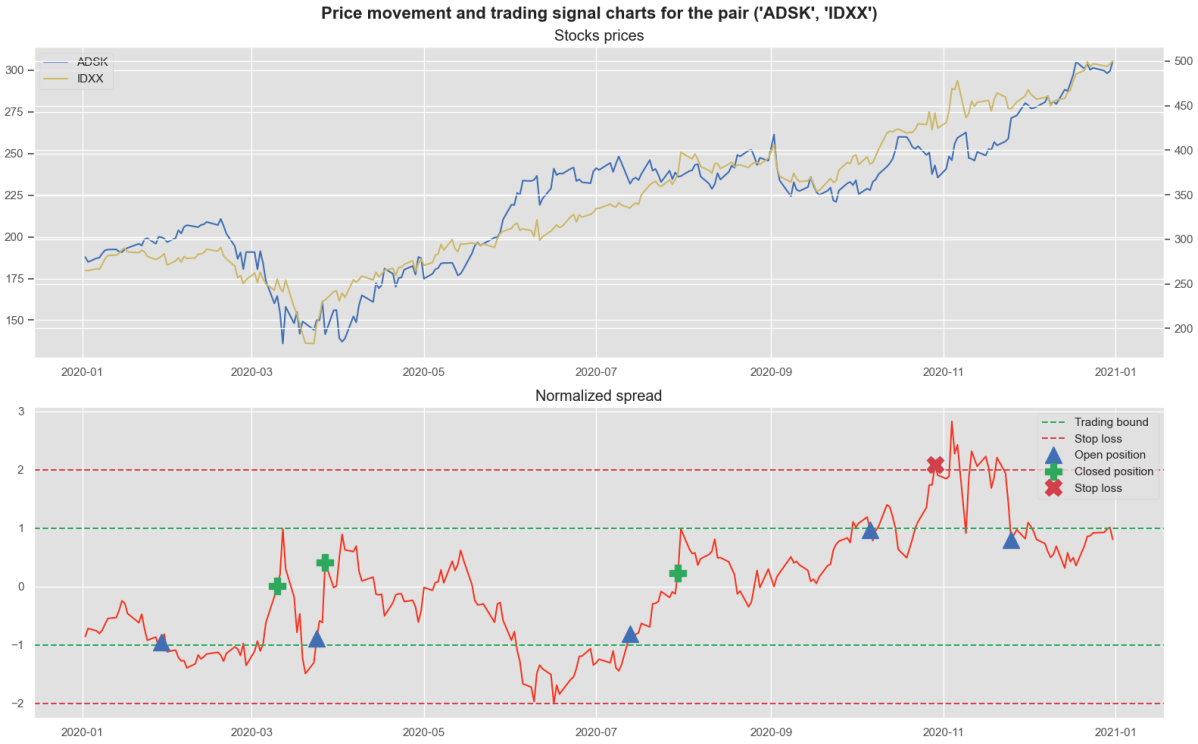

Figure 17: Price movement chart and trading signal generated for "ADSK", "IDXX"

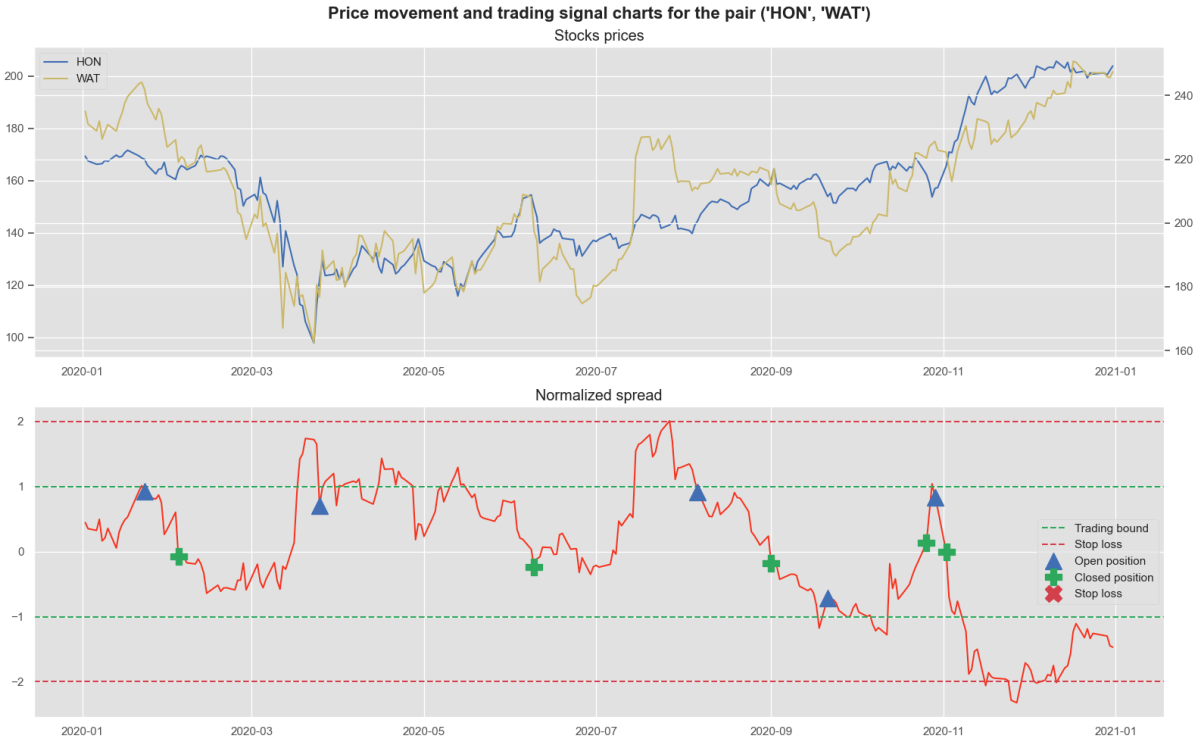

Figure 18: Price movement chart and trading signal generated for ”HON”, ”WAT”

References

1. Göncü, A. and Akyıldırım, E. (2016). Statistical Arbitrage with Pairs Trading. International Review of Finance, 16(2), pp.307–319. doi:https://doi.org/10.1111/irfi.12074

2. Krauss, C. (2016). STATISTICAL ARBITRAGE PAIRS TRADING STRATEGIES: REVIEW AND OUTLOOK. Journal of Economic Surveys, 31(2), pp.513–545. doi:https://doi.org/10.1111/joes.12153.

3. Qazi, L.T., Rahman, A.U. and Gul, S. (2015). Which Pairs of Stocks should we Trade? Selection of Pairs for Statistical Arbitrage and Pairs Trading in Karachi Stock Exchange. The Pakistan Development Review, [online] 54(3), pp.215–244. Available at: http://www.jstor.org/stable/43830729 [Accessed 21 Apr. 2023].

4. Lu, J.-Y., Lai, H.-C., Shih, W.-Y., Chen, Y.-F., Huang, S.-H., Chang, H.-H., Wang, J.-Z., Huang, J.-L. and Dai, T.-S. (2021). Structural break-aware pairs trading strategy using deep reinforcement learning. The Journal of Supercomputing, 3843–3882 (2022)(2022). doi:https://doi.org/10.1007/s11227-021-04013-x

5. Ko, H. (2020). Cluster analysis on stock selection. [online] Medium. Available at: https://towardsdatascience.com/clustering-analysis-on-stock-selection-2c2fd079b295.

6. Barziy, I. (2021). Machine Learning for Trading Pairs Selection. [online] Hudson & Thames. Available at: https://hudsonthames.org/employing-machine-learning-for-trading-pairs-selection/ [Accessed 18 Apr. 2023].

7. Bin, S. (2020). K-Means Stock Clustering Analysis Based on Historical Price Movements and Financial Ratios. CMC Senior Theses, [online] 2020(2435). Available at: https://scholarship.claremont.edu/cmc_theses/2435 [Accessed 18 Apr. 2023].

Group 6: Contributions

Name	Contributions
Kuntoji, Rohan	• Research on various Machine Learning and NLP techniques for clustering and stock selection • Code implementation & visualisations
Parary, Aman	• Research on data gathering, DB integration and data manipulation strategies • Code implementation and modularisation
Fernandes, William	• Research on statistical arbitrage strategy • Literature survey on stock clustering methods and approaches

Python Code

```
#!/usr/bin/env python
# coding: utf-8
#PATH: ./main.py
#--This is the main file that runs all the functions from
# data_generation, stock_clustering, pairs_trading

from utils.data_generation import *
from utils.stock_clustering import *
from utils.pairs_trading import *
import sqlite3
import pandas as pd

## -- Novel Data Set Collection -- ##

# Initialise the database
conn = sqlite3.connect('data/pairs_trading.db')

sp500_comp_profile_df = fetch_sp500_comp()

ticker_list = sp500_comp_profile_df['ticker'].to_list()
sp500_comp_profile_df = get_bus_desc_data(tickers=ticker_list, conn=conn)
stocks_hist_price_df = get_hist_price_data(tickers=ticker_list,
start_date="2010-01-01", end_date="2023-03-31", conn=conn)
stocks_hist_ratios_df = get_hist_ratios_data(ticker_list, start=2010, conn=conn)

## -- End of Novel Data Set Collection -- ##


## -- Database Querying and Reporting -- ##

display_analysis(conn)


## -- End of Database Querying and Reporting -- #

## -- Data Preparation -- ##

# clean and preprocess the stock ratios data

stock_ratios_df = pd.read_sql_query("SELECT * FROM stocks_hist_ratios", conn)
sp500_stocks_profile_df = pd.read_sql_query("SELECT * FROM stocks_profile", conn)

# clean and preprocess ratios data
target_ratios = ['Quick ratio', 'Cash ratio', 'Interest coverage', 'Debt equity ratio',
'Asset turnover', 'Receivables turnover', 'Return on assets', 'Operating profit margin',
'Enterprise value multiple', 'Payout ratio']

ratios_pp_df = clean_preprocess_ratios_data(stock_ratios_df)

# clean and preprocess stock business descriptions text
document_topic_df, word_topic_df, sing_topic_df =
preprocess_bus_desc_data(sp500_stocks_profile_df[['ticker', 'business_desc']],n_topics=80)

# Plot top 10 terms within each topic
plot_topic_top10_terms(word_topic_df)
```

```
# combined feature space creation
final_features_df = combine_and_normalize_data(ratios_pp_df, document_topic_df)
print(final_features_df.shape)

## -- End of Data Preparation -- ##

## -- Model Building -- ##

# K-Means
km_cluster_df = find_and_fit_optimal_kmeans(final_features_df)

# OPTICS
db_cluster_df = optics_fit(final_features_df, min_samples=7)

# Hierarchical
hc_cluster_df = agg_hc_fit(final_features_df, n_clusters=14, linkage='average')

# Cluster Analysis
compare_clustering_results(km_cluster_df, db_cluster_df, hc_cluster_df, sp500_stocks_profile_df)
final_cluster_df = get_final_clabel_profile_df(db_cluster_df, target_ratios)
boxplot_cluster_fin_ratios(final_cluster_df, target_ratios)
plot_cluster_wordclouds(final_cluster_df)

## -- End of Model Building -- ##

## -- Textual Analysis -- ##

filtered_stock_profiles = filter_stock_profiles(sp500_stocks_profile_df)
filtered_stock_profiles = apply_preprocessing(filtered_stock_profiles)
unique_words_count = count_unique_words(filtered_stock_profiles)
tfidf_sparse_matrix = tfidf_transform(filtered_stock_profiles)
visualize_sparse_matrix(tfidf_sparse_matrix)

## -- End of Textual Analysis -- ##

##-- Pairs Trading --

# find cointegrated pairs and perform pairs selection
hist_final_stock_pairs = cluster_pair_selection(cluster_df=target_cluster_df)
print(f"Number of clusters: {len(target_cluster_df.cluster.value_counts())}")
print(f"Number of cointegrated pairs: {len(hist_final_stock_pairs)}")
print(f"Pairs with lowest p-value from each clusters:\n{hist_final_stock_pairs}")

# create a portfolio of selected pairs based on the set criterion
portfolio = Portfolio(stocks_df=full_hist_close_df, pairs_list=hist_final_stock_pairs)
print(f'Selected pairs: \n {portfolio.selected_pairs}')
portfolio.plot_portfolio()

# plot performance charts for the benchmark for comparison
plot_benchmark_ret(conn)

## -- End of Pairs Trading --
conn.close()
```

```
#!/usr/bin/env python
# coding: utf-8
#PATH: utils/data_generation.py
# In[9]:


import os
from dotenv import load_dotenv
load_dotenv()

import pandas as pd
import numpy as np
import datetime as dt
from collections import Counter
pd.set_option('display.max_rows', 2000)
pd.set_option('display.max_columns', 1000)
pd.set_option('display.width', 1000)

from tqdm import tqdm
from openbb_terminal.sdk import openbb
openbb.keys.fred(key=os.getenv('FRED_API_KEY_1'))
openbb.keys.fmp(key=os.getenv('FMP_RSK_KEY_2'))
import sqlite3
import requests
import time

import warnings
warnings.filterwarnings("ignore")

from typing import List, Tuple


# Pull the S&P 500 constituents profile data from Wiki page (public data)
def fetch_sp500_comp(wiki_url: str=os.getenv('WIKI_SP500_URL')):
    """
    Fetch the S&P 500 constituents profile data from Wiki page (public data)
    """
    wiki_table = pd.read_html(wiki_url)
    sp500_comp_profile = wiki_table[0]
    sp500_comp_profile['Symbol'] = sp500_comp_profile['Symbol'].str.replace('\\.', '-', regex=True)
    sp500_comp_profile.rename(columns={"Symbol": "ticker", "Security": "company_name",
    "CIK": "cik", "GICS Sector": "sector", "GICS Sub-Industry": "sub_industry",
    "Headquarters Location": "hq", "Founded": "founded",
                                       "Date added": "date_added"}, inplace=True)
    return sp500_comp_profile

def get_bus_desc_data(tickers, conn):
    """
    Returns a Series of bus descriptions.
    """
    # Initialize an empty dict for the ticker : description data
    bus_desc = {}
    # Loop over each ticker symbol and get the company description
    for tkr in tqdm(tickers):
        # Define the API endpoint with the ticker symbol and API key
        API_KEY = os.getenv('ALPHA_VANT_API_KEY')
        endpoint = f'https://www.alphavantage.co/query?function=OVERVIEW&symbol={tkr}&apikey={API_KE
```

```
        print(f"Processing ticker: {tkr}")  # Added to show the ticker being processed

        # Make the API request and extract the company description from the response
        try:
            key_stats_response = requests.request("GET", endpoint)
            response_data = key_stats_response.json()

            if 'Description' in response_data:
                bus_desc[tkr] = response_data['Description']
            else:
                print(f"No 'Description' field found in response for {tkr}")
                bus_desc[tkr] = None

            # print(response_data)
            print(f'{bus_desc}')

            # Wait for 12 seconds before the next API call to stay within the rate limit of 5 calls
            time.sleep(12)

        except Exception as e:
            print(f"Error for {tkr}: {e}")
            break

    bus_desc_df = pd.Series(bus_desc)
    bus_desc_df = bus_desc_df.reset_index(name='business_desc').rename(columns={'index': 'ticker'})
    bus_desc_df.to_sql('stocks_profile', conn, if_exists='replace', index=False)
    return bus_desc_df


def get_hist_price_data(tickers, start_date: str, end_date: str, conn):
    """
    Get historical price data from OpenBB python package - AlphaVantage
    start_date: str="2010-01-01"
    end_date: str="2023-03-31"

    """
    stock_dict = {}
    for tkr in tqdm(tickers):
        stock_data = openbb.stocks.load(tkr, start_date=start_date, end_date=end_date,
        source='AlphaVantage',
        verbose=False).reset_index().rename(columns={"date": "date"})

        if len(stock_data) > 0:
            # calculate simple returns & log returns
            stock_data['simple_return'] = stock_data['Adj Close'].pct_change()
            stock_data['log_return'] = np.log(stock_data['Adj Close']/stock_data['Adj Close'].shift(
        else:
            print(f'No data found for {tkr}')

        stock_dict[tkr] = stock_data
        time.sleep(11) # wait for 11 seconds before the next API call to stay within the rate limit

    # create a list of DFs of individual stock historical price data
    dfs = []
    for ticker, data in stock_dict.items():
        data['ticker'] = ticker
        dfs.append(data)
```

```
    # concatenate all DFs to form a single large DF
    all_stock_hist_price = pd.concat(dfs, ignore_index=False)
    # all_stock_hist_price = all_stock_hist_price.drop('index', axis=1)
    all_stock_hist_price['date'] = pd.to_datetime(all_stock_hist_price['date'])
    # set the order of columns for better readability
    column_order = ['ticker'] + [col for col in all_stock_hist_price.columns if col != 'ticker']
    all_stock_hist_price = all_stock_hist_price[column_order]

    # Check if any stocks were left out
    tkr_unique = all_stock_hist_price['ticker'].unique()
    tkr_dropped = [ tkr for tkr in tickers if tkr not in tkr_unique ]
    print(f'{len(tkr_dropped)} missed out: {tkr_dropped}') # none were dropped, awesome!

    # Create a separate DF containing historical price data of all the stocks
    #and store it to the database
    all_stock_hist_price.to_sql('stocks_hist_price', conn, if_exists='replace', index=False)
    return all_stock_hist_price

def get_hist_ratios_data(tickers, start: int, conn):
    """
    Get historical financial ratios data from OpenBB python package - Financial Modeling Prep
    start: int=2010
    """
    ratios_dict = {}
    for ticker in tqdm(tickers):
        stock_ratios = openbb.stocks.fa.ratios(ticker, 15)
        if len(stock_ratios) > 0:
            stock_ratios = stock_ratios.T
            stock_ratios = stock_ratios.drop(columns='Period').reset_index()
            stock_ratios['Fiscal Date Ending'] = stock_ratios['Fiscal Date Ending'].astype(int)
            stock_ratios = stock_ratios[stock_ratios['Fiscal Date Ending'] >= 2010]
            stock_ratios = stock_ratios.reindex(index=stock_ratios.index[::-1]
            ).reset_index(drop=True)
            stock_ratios['ticker'] = ticker
        else:
            print(f'No data found for {ticker}')
        ratios_dict[ticker] = stock_ratios

    # create a list of DFs of individual stock historical ratios data
    ratio_dfs = []
    for tkr, data in ratios_dict.items():
        ratio_dfs.append(data)

    # concatenate all DFs to form a single large DF
    all_stock_hist_ratios = pd.concat(ratio_dfs, ignore_index=False)
    # all_stock_hist_ratios = all_stock_hist_price.drop('index', axis=1)
    # set the order of columns for better readability
    column_order = ['ticker'] + [col for col in all_stock_hist_ratios.columns if col != 'ticker']
    all_stock_hist_ratios = all_stock_hist_ratios[column_order]

    # Check if any stocks were left out
    tkr_unique = all_stock_hist_ratios['ticker'].unique()
    tkr_dropped = [ tkr for tkr in tickers if tkr not in tkr_unique ]
    print(f'{len(tkr_dropped)} missed out: {tkr_dropped}') # 1 stock missed out ('PEG')
    # For now, let's not re-fetch the price data for dropped tickers,
    #we'll proceed with what we have

    # Create a separate DF containing historical price data of all the stocks
```

```
    #and store to the database
    all_stock_hist_ratios.to_sql('stocks_hist_ratios', conn, if_exists='replace', index=False)
    return all_stock_hist_ratios


def display_analysis(conn):
    stocks_hist_price_df = pd.read_sql_query("SELECT * FROM stocks_hist_price", conn)
    print(f'Total count of unique stocks: {len(stocks_hist_price_df.ticker.unique())}\n')

    stocks_hist_ratios_df = pd.read_sql_query("SELECT * FROM stocks_hist_ratios", conn)
    print(f'Total count of unique stocks: {len(stocks_hist_ratios_df.ticker.unique())}\n')

    sp500_comp_profile_df = pd.read_sql_query("SELECT * FROM stocks_profile", conn)

    profile_tickers = sp500_comp_profile_df.ticker.unique().tolist()
    print(f'Tickers in profile dataset ({len(profile_tickers)}):\n{profile_tickers}')
    print('--'*20)
    price_tickers = stocks_hist_price_df.ticker.unique().tolist()
    print(f'Tickers in price dataset ({len(price_tickers)}):\n{price_tickers}')
    print('--'*20)
    ratio_tickers = stocks_hist_ratios_df.ticker.unique().tolist()
    print(f'Tickers in ratios dataset ({len(ratio_tickers)}):\n{ratio_tickers}')
    print('--'*20)
    common_tickers_universe = [tkr for tkr in profile_tickers if
    tkr in price_tickers and tkr in ratio_tickers]

    print(f'\nCommon universe of tickers ({len(common_tickers_universe)}):
    \n{common_tickers_universe}')

    sector_counts = pd.read_sql_query('''SELECT sector, COUNT(*) as num_companies
                                         FROM stocks_profile
                                         GROUP BY sector
                                         ORDER BY num_companies DESC
                                         LIMIT 10''', conn)

    print("Top 10 sectors with the highest number of companies")
    print(sector_counts)

    companies_by_location = pd.read_sql_query('''
    SELECT
    -- extract the state or country from the 'hq' column
        CASE
            -- Check if the 'hq' value contains a comma followed by a space (', ')
            WHEN hq LIKE '%, %' THEN
                -- If there's a comma followed by a space, extract the string after the comma
                SUBSTR(hq, INSTR(hq, ',') + 2)
                -- If there's no substring after a comma, then set as Unknown
            ELSE 'Unknown'
        END as location,
        COUNT(*) as num_companies
    FROM stocks_profile
    GROUP BY location
    ORDER BY num_companies DESC
    LIMIT 10
    ''', conn)

    print("\nNumber of companies based on each location:")
    print(companies_by_location)
```

```
top_10_avg_volume = pd.read_sql_query('''
SELECT ticker, printf('%.0f', AVG(Volume)) as avg_volume
GROUP BY ticker
ORDER BY avg_volume DESC
LIMIT 10
''', conn)

print("\nTop 10 tickers with the highest average daily trading volume:")
print(top_10_avg_volume)

top_10_days_total_volume = pd.read_sql_query(
'''
SELECT strftime('%Y-%m-%d', date) as date,
printf('%.0f', SUM(Volume)) as total_volume
FROM stocks_hist_price
GROUP BY date
ORDER BY total_volume DESC
LIMIT 10
''',conn)

print("\nTop 10 trading days with the highest total trading volume across all tickers:")
print(top_10_days_total_volume)
top_10_roe_companies = pd.read_sql_query("""
SELECT ticker, MAX("Fiscal Date Ending") as latest_fiscal_date,
MAX("Return on equity") as ROE FROM stocks_hist_ratios
GROUP BY ticker
ORDER BY ROE DESC
LIMIT 10""",
conn)

print("\nTop 10 companies with the highest Return on Equity (ROE) for 2022:")
print(top_10_roe_companies)


top_10_gpm_companies = pd.read_sql_query(
"""
SELECT ticker, MAX("Fiscal Date Ending") as latest_fiscal_date,
MAX("Current ratio") as current_ratio
FROM stocks_hist_ratios
GROUP BY ticker
ORDER BY current_ratio DESC
LIMIT 10""",
conn)

print("\nTop 10 companies with the highest Current Ratio for 2022:")
print(top_10_gpm_companies)
```

```
#!/usr/bin/env python
# coding: utf-8
#PATH: utils/stock_clustering.py
##--This script contains code for data pre-processing, data visualisation, text analytics, and model

import pandas as pd
import numpy as np
import datetime as dt
from collections import Counter
pd.set_option('display.max_rows', 2000)
pd.set_option('display.max_columns', 1000)
pd.set_option('display.width', 1000)
import sqlite3
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
import colorcet as cc
sns.set(style="white", rc={"figure.figsize":(8, 4)})
plt.style.use('ggplot') # fivethirtyeight, ggplot, dark_background, classic,

from sklearn.preprocessing import StandardScaler, Normalizer, MinMaxScaler
from sklearn import feature_extraction
from sklearn.decomposition import TruncatedSVD, PCA
from sklearn.cluster import KMeans, OPTICS, AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage, cophenet
from sklearn.metrics import silhouette_score, calinski_harabasz_score
from sklearn.manifold import TSNE
from sklearn.impute import KNNImputer
from scipy.spatial.distance import pdist, cdist

from kneed import KneeLocator
import re
import nltk
# nltk.download('stopwords') # required to be downloaded only for the first time
# nltk.download('wordnet') # required to be downloaded only for the first time
# nltk.download('omw-1.4') # required to be downloaded only for the first time
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
from wordcloud import WordCloud

import warnings
warnings.filterwarnings("ignore")


def std_normalise_data(data = pd.DataFrame()):
    """
    Standard scaling and Normalisation of input data

    Parameters:
    -----------
    data : pd.DataFrame
            raw data

    Returns:
    --------
    new_df : pd.DataFrame
            scaled data
    """
```

```
    new_data = StandardScaler().fit_transform(data)
    new_data = Normalizer().fit_transform(new_data)
    new_df = pd.DataFrame(new_data, index=data.index, columns=data.columns)
    return new_df

def knn_imputer(data: pd.DataFrame, n_neighbours: int=30):
    """
    Data imputation with KNNImputer

    Parameters:
    -----------
    data : pd.DataFrame
            raw data that needs to be imputed to handle missing values.

    n_neighbours : int, default value is 30
            Number of neighboring samples to use for imputation.

    Returns:
    --------
    new_df : pd.DataFrame
            imputed data with zero NaN values.
    """
    # scale the data using minmaxscaler
    scaled_data = MinMaxScaler(feature_range=(0, 1)).fit_transform(data)
    # Define KNN imputer and fill missing values
    knn_fit_data = KNNImputer(n_neighbors=n_neighbours, weights='distance'
    ).fit_transform(scaled_data)

    new_df = pd.DataFrame(knn_fit_data, columns=data.columns, index=data.index)

    # check if there are any more missing values after imputation
    print(f'Total count of missing values before imputing with KNNImputer: {data.isna().sum().sum()
    }')
    print(f'Total count of missing values after imputing with KNNImputer:{new_df.isna().sum().sum()
    }')
    return new_df

def pca_tuning(data = pd.DataFrame()):
    """
    PCA tuning - dimensionality reduction

    method 1 - Plot the cummulative explained variance against the count of principal components

    method 2 - Create a 'Scree' plot which gives the visual representation of eigenvalues
    that define the magnitude of eigenvectors (principal components)

    Parameters:
    -----------
    data : pd.DataFrame
            data on which to fit the PCA model

    Returns:
    --------
    pca_base : PCA object
            Fitted PCA model object

    knee_value : int
            computed Knee point from the Scree plot
```

```
    """
    # Start with basic PCA, keeping all components with no reduction
    pca_base = PCA()
    pca_base.fit(data)

    plt.figure(figsize=(12, 6))
    # Method 1
    # Fetch variance explained by each individual component & compute the cummulative sum
    print('Method 1 - variance explained by components')
    plt.subplot(1, 2, 1)
    exp_var = pca_base.explained_variance_ratio_ * 100
    cum_exp_var = np.cumsum(exp_var)

    plt.bar(range(0, exp_var.size), exp_var, align='center',
    label='Individual explained variance (%)')

    plt.step(range(0, cum_exp_var.size), cum_exp_var, where='mid',
    label='Cumulative explained variance (%)', color='indigo')

    plt.ylabel('Explained variance (%)')
    plt.xlabel('Principal component index')
    plt.legend(loc='best')

    # Method 2 - Scree plot
    print('\nMethod 2 - Scree plot')
    plt.subplot(1, 2, 2)
    plt.plot(np.arange(pca_base.n_components_) + 1, pca_base.explained_variance_, 'ro-',
    linewidth=2,  color='orange')
    knee = KneeLocator(np.arange(pca_base.n_components_) + 1, pca_base.explained_variance_, S=20,
                       curve='convex', direction='decreasing', interp_method='interp1d')
    plt.axvline(x=knee.knee, ymax=0.95, ymin=0.05, color='brown', ls='--')
    plt.figtext(0.65, 0.4, f'optimal n_comp = {knee.knee}', bbox=dict(facecolor='violet', alpha=0.3)
    plt.title("Scree Plot")
    plt.xlabel("Principal component index")
    plt.ylabel("Eigenvalues")
    print(f'Knee point, eigenvalue : {knee.knee, pca_base.explained_variance_[knee.knee]}')
    plt.tight_layout(pad=2.0)
    plt.show()

    return pca_base, knee.knee


def clean_preprocess_ratios_data(raw_data: pd.DataFrame, target_date_range: range=range(2010, 2020),
                                 target_ratios: list=None):
    """
    Perform data cleaning and preprocessing
    on raw historical financial ratios dataset.

    Parameters:
    -----------
    raw_data : pd.DataFrame
        large single dataframe of historical ratios for all the stocks.

    target_date_range : range
        range of fiscal years to be considered for fetching annual historical ratios measures.
        It's a range of years which includes the first value but not the last value.
```

```
    copy : bool, default value is True
        If set to True, all processing will be done on a copy of the input raw data.
        Else, the input raw data will be modified.

    target_ratios : list, default list of desired targets defined
        The list of specific ratios to be considered as the target feature set for model fitting.

    Returns:
    --------
    ratios_final_df : pd.DataFrame
        cleaned and processed dataframe of target historical ratio measures
        within the specified date range.
    """

    if target_ratios is None:
        target_ratios = ['Quick ratio', 'Cash ratio', 'Interest coverage', 'Debt equity ratio',
        'Asset turnover', 'Receivables turnover', 'Return on assets', 'Operating profit margin', 'En


    if copy:
        raw_data = raw_data.copy()
    # reshuffling DF into a pivot DF
    pivot_df = raw_data[['ticker', 'Fiscal Date Ending'] + target_ratios].pivot(
        index='ticker', columns='Fiscal Date Ending', values=target_ratios)
    # change the dtypes from object to float
    pivot_df = pivot_df.apply(pd.to_numeric, errors='coerce', downcast='float')
    # filter data only within the target date range
    target_df = pivot_df[[col_tup for col_tup in pivot_df.columns
                                if col_tup[1] in target_date_range]]

    # data imputation of missing values
    # From our EDA, we found KNNImputer works the best on this data for n_neighbours=30
    print('-'*50)
    print('Data Imputation\n')
    target_imp_df = knn_imputer(target_df, n_neighbours=30)

    # PCA - dimensionality reduction
    # standard scaling and normalisation before fitting PCA
    print('-'*50)
    print('PCA for Dimensionality Reduction\n')
    scaled_df = std_normalise_data(target_imp_df)
    # Tune a default PCA model and find optimal n_components for fitting a PCA
    # Note: PCA needs the input data to be in (n_features, n_samples) format
    # for it to function properly and reduce the feature dimensionality
    pca_base, knee_value = pca_tuning(scaled_df.T)
    print('-'*50)
    print(f'Fit a new PCA with n_components = {knee_value} as observed from the Scree plot')
    # Fit a new PCA model with the obtained hyperparam value testing
    pca_final = PCA(n_components=knee_value)
    # Note: we need to input data in the exact required format here for the PCA to work properly
    pca_final.fit(scaled_df.T)

    # Structure the final results in a dataframe
    ratios_final_df = pd.DataFrame(pca_final.components_.T, index=scaled_df.index,
    columns=[f'PC_{i}' for i in range(pca_final.components_.T.shape[1])])

    return ratios_final_df
```

```
def text_cleaning(text: str, flg_stemm=False, flg_lemm=True):
    """
    Clean & preprocess input string data. (remove stop words from the text, stemming/lemmatisation)

    Parameters:
    -----------
    text : str
        Textual data of string type.

    flg_stemm : bool, default is False
        Flag to indicate whether stemming should be performed on the input text.
        Note: You should not set both flg_stemm & flg_lemm to be True. Only one of them can be True

    flg_lemm : bool, default value is True
        Flag to indicate whether lemmatisation should be performed on the input text.
        Note: You should not set both flg_stemm & flg_lemm to be True. Only one of them can be True

    Returns:
    --------
    text : str
        cleaned and processed string (removed stop words, stemming/lemmatisation).
    """
    # clean (convert to lowercase and remove punctuations and special characters and then strip)
    text = re.sub(r'[^\w\s]', '', str(text).lower().strip())
    # Tokenize (convert from string to list)
    tokens = text.split()
    # remove Stopwords
    # extend the default list by adding more non-important words
    stop_words.extend(['founded', 'firm', 'company', 'llc', 'inc', 'incorporated',
                       'multinational', 'corporation', 'commonly', 'headquartered'])
    if stop_words is not None:
        tokens = [word for word in tokens if word not in stop_words]
    # Stemming (remove -ing, -ly, ...)
    if flg_stemm == True:
        ps = nltk.stem.porter.PorterStemmer()
        tokens = [ps.stem(word) for word in tokens]
    # Lemmatisation (convert the word into root word)
    if flg_lemm == True:
        lem = nltk.stem.wordnet.WordNetLemmatizer()
        tokens = [lem.lemmatize(word) for word in tokens]

    # back to string from list
    text = ' '.join(tokens)
    return text


def preprocess_bus_desc_data(raw_data: pd.DataFrame, copy: bool=True, n_topics: int=150):
    """
    Perform data cleaning and preprocessing
    on raw historical financial ratios dataset.

    Parameters:
    -----------
    raw_data : pd.DataFrame
        Dataframe containing stock tickers and their respective business descriptions.

    copy : bool, default value is True
        If set to True, all processing will be done on a copy of the input raw data.
```

```
    Else, the input raw data will be modified.

n_topics : int, default value is 150
    number of topics to be deduced from LSA.

Returns:
--------
document_topic_df : pd.DataFrame
    DF containing Document-topic matrix values.

word_topic_df : pd.DataFrame
    DF containing Word-topic matrix values.

sing_topic_df : pd.DataFrame
    DF containing singular matrix values.
"""

if copy:
    raw_data = raw_data.copy()

# drop tickers with missing values in business_desc column only
print('-'*50)
print(f'Drop stocks with missing business description data since this forms the bedrock'
f'of feature creation\n')

raw_data.dropna(axis=0, subset=['business_desc'], inplace=True)
print(f'Total stocks in consideration after dropping the ones with missing business
descriptions: {raw_data.shape[0]}')
# perform text cleaning to get rid of stopwords, apply stemming/lemmatisation rules
print('-'*50)
print('Carry out text cleaning which includes - removing stopwords, stemming & lemmatisation\n')
raw_data['bd_clean'] = raw_data['business_desc'].apply(lambda txt: text_cleaning(txt))
print(f'Total count of unique words (unigrams) found in our corpus of business descriptions'
f'across all stocks: {len(set([ele for arr in list(raw_data["bd_clean"].apply(
lambda x: str(x).split(" "))) for ele in arr]))}')

# Vectorise the textual data using TF-IDF method and get a sparse
#DTM on the cleaned data (tuning the Tf-Idf hyperparams min_df, max_df & ngram_range)
print('-'*50)
print('Vectorise cleaned text data using TF-IDF method, get a sparse DTM '
f'- improve vectorisation with appropriate hyperparam values\n')
tfidf = feature_extraction.text.TfidfVectorizer(stop_words='english', max_features=10000,
min_df=1, max_df=0.1, strip_accents='unicode', ngram_range=(2,4))
tfidf_sparse = tfidf.fit_transform(raw_data['bd_clean'])
tfidf_features = tfidf.get_feature_names_out().tolist()
# let's see some features which were extracted by Tf-Idf
print(f'TF-IDF feature names (a few samples): \n{tfidf_features[-100:]}\n')

# Apply LSA to reduce dimensionality of TF-IDF sparse matrix
print('-'*50)
print('Apply LSA to reduce dimensionality of TF-IDF sparse matrix\n')
# define a desired n_components (themes/topics) for a Truncated SVD model
lsa_obj = TruncatedSVD(n_components=n_topics, n_iter=100, random_state=1500)
# define tfidf sparse matrix as an actual DF
tfidf_sparse_df = pd.DataFrame(data=tfidf_sparse.toarray(), index=raw_data.ticker,
columns=tfidf_features)
# compute document-topic matrix
document_topic_m = lsa_obj.fit_transform(tfidf_sparse_df)
```

```
    # compute word-topic matrix
    word_topic_m = lsa_obj.components_.T
    # compute singular values topic matrix
    sing_topic_m = lsa_obj.singular_values_
    print(f'Reduced dimensionality from {len(tfidf_features)} to {lsa_obj.components_.shape[0]}\n')
    print(f'Total variance explained by the top {lsa_obj.components_.shape[0]} topics (%):
    # define topics
    topics = [f'Topic_{i}' for i in range(0, sing_topic_m.shape[0])]
    document_topic_df = pd.DataFrame(data=document_topic_m, index=raw_data.ticker, columns=topics)
    word_topic_df = pd.DataFrame(data=word_topic_m, index=tfidf_sparse_df.columns, columns=topics)
    sing_topic_df = pd.DataFrame(data=sing_topic_m, index=topics)

    return document_topic_df, word_topic_df, sing_topic_df


def plot_topic_top10_terms(data_df: pd.DataFrame):
    """
    Plot top 10 terms (ngram) within the topic for
    a sample of topics.

    Parameters:
    -----------
    data_df : pd.DataFrame
        Word-topic DF computed from the LSA algoithm.
    """
    print('-'*50)
    print('Plot top terms (ngrams) within each topic\n')
    fig1 = plt.figure(figsize=(20, 20))
    fig1.subplots_adjust(hspace=.5, wspace=.5)
    plt.clf()
    for i in range(0, 6):
        fig1.add_subplot(5, 2, i+1)
        temp = data_df.iloc[:, i*15]
        temp = temp.sort_values(ascending=False)
        plt.title(f'Top 10 terms (ngrams) in Topic {i*15}', weight='bold', fontsize=14)
        sns.barplot(x= temp.iloc[:10].values, y=temp.iloc[:10].index)
        i += 1
    fig1.tight_layout(pad=2.0)
    plt.show()

def combine_and_normalize_data(ratios_pp_df: pd.DataFrame, document_topic_df: pd.DataFrame,
normalise_function=std_normalise_data):
    """
    Combines the ratios_pp_df and document_topic_df dataframes, and applies standard scaling
    and normalization.
    """
    final_features_df = pd.concat([ratios_pp_df, document_topic_df], axis=1, join='inner')

    # Apply standard scaling and normalization
    final_features_df = normalise_function(final_features_df)

    fig, ax = plt.subplots(figsize=(8,8))
    plt.title('Feature space correlation map\n', fontsize=16)
    sns.heatmap(final_features_df.corr(), ax=ax)
    plt.show()

    return final_features_df
```

```
def plot_TSNE(data: pd.DataFrame, labels):
    """
    Plot the results of the t-SNE algorithm to visualise the
    cluster formations from high dimensional data on a 2-D plot

    Parameters:
    -----------
    data : pd.DataFrame
        Dataframe containing processed data on which the clustering model was fit.

    labels : nd array
        Array of computed cluster labels.

    Returns:
    --------
    clustered_series_all : pd.Series
        Pandas series with tickers and cluster labels.
    """
    # all stock with its cluster label (including -1)
    clustered_series_all = pd.Series(index=data.index, data=labels)
    # use TSNE algorithm to plot multidimension into 2D
    tsne_data = TSNE(n_components=2, perplexity=80, random_state=1337).fit_transform(data)
    tsne_df = pd.DataFrame(data=tsne_data, index=data.index, columns=['tsne_1', 'tsne_2'])
    tsne_df['label'] = labels
    # clustered
    tsne_df = tsne_df[tsne_df['label'] != -1]
    sns.lmplot(data=tsne_df, x='tsne_1', y='tsne_2', hue='label', height=6, aspect=2,
    fit_reg=False, legend=True, palette=sns.color_palette(cc.glasbey,
    len(tsne_df.label.value_counts())), scatter_kws={'s':150, 'alpha': 0.5})
    # unclustered in the background
    plt.scatter(tsne_data[(clustered_series_all==-1).values, 0],
    tsne_data[(clustered_series_all==-1).values, 1], s=50, alpha=0.05)
    plt.title('t-SNE plot for all stocks with cluster labels\n', weight='bold').set_fontsize('14')
    plt.xlabel('t-SNE comp 1', weight='bold', fontsize=12)
    plt.ylabel('t-SNE comp 2', weight='bold', fontsize=12)
    plt.tight_layout(pad=2.0)
    plt.show()
    return clustered_series_all


# show number of stocks in each cluster
def plot_cluster_counts(labels_df: pd.DataFrame):
    """
    Plot cluster counts histogram bar chart

    Parameters:
    -----------
    labels_df : pd.DataFrame (Series)
        Pandas Series containing tickers and cluster labels as output by plot_TSNE.
    """
    plt.figure(figsize=(12,8))
    labels_df[labels_df!=-1].value_counts().sort_index().plot(kind='barh')
    plt.title('Cluster stock counts\n', weight='bold', fontsize=12)
    plt.xlabel('Stocks in Cluster', weight='bold', fontsize=10)
    plt.ylabel('Cluster Number', weight='bold', fontsize=10)
    plt.tight_layout(pad=2.0)
    plt.show()
```

```
# plot price movements for cluster stocks
def plot_cluster_members(labels_df: pd.DataFrame):
    """
    Plot the cluster members' (stocks) log prices
    to observe if there's any similarity in patterns

    Parameters:
    -----------
    labels_df : pd.DataFrame (Series)
        Pandas Series containing tickers and cluster labels as output by plot_TSNE.
    """
    # get the number of stocks in each cluster
    conn = sqlite3.connect('./data/pairs_trading.db')
    counts = labels_df[labels_df!=-1].value_counts()
    # let's visualize some clusters
    cluster_vis_list = list(counts[counts>1].sort_values().index)
    # this code needs to be replaced with code to fetch from db
    hist_price_df = pd.read_sql_query("SELECT * FROM stocks_hist_price", conn, parse_dates=['date'])
    #pd.read_csv('./data/stocks_hist_price.csv', date_parser=['date'])
    hist_price_df.date = pd.to_datetime(hist_price_df.date, format='ISO8601')
    sp500_stocks_profile_df = pd.read_sql_query("SELECT * FROM stocks_profile", conn)
    conn.close()
    temp_df = hist_price_df.pivot_table(values='Adj Close', index='date', columns='ticker')
    # plot a handful of the smallest clusters
    plt.figure(figsize=(24, 48))
    plt.subplots_adjust(hspace=.25, wspace=.25)
    plt.clf()
    i=1
    for clust in cluster_vis_list[0:min(len(cluster_vis_list), 4)]:
        tickers = list(labels_df[labels_df==clust].index)
        means = np.log(temp_df.loc[:dt.datetime(2019, 12, 31), tickers].mean())
        data = np.log(temp_df.loc[:dt.datetime(2019, 12, 31), tickers]).sub(means)
        plt.subplot(4, 1, i)
        plt.plot(temp_df.loc[:dt.datetime(2019, 12, 31), tickers].index, data)
        plt.title(f'Time series of log returns for stocks in Cluster {clust}\n',
        weight='bold', fontsize=14)
        plt.xlabel('Date', weight='bold', fontsize=12)
        plt.ylabel('Log returns', weight='bold', fontsize=12)
        tkr_names = [f'{tkr} - {sp500_stocks_profile_df[sp500_stocks_profile_df.ticker ==
        tkr].company_name.values[0]}' for tkr in tickers]
        plt.legend(tkr_names)
        i+=1
    plt.tight_layout(pad=2.0)
    plt.show()


def find_optimal_k(data = pd.DataFrame(), k_range: range=range(2, 50), init: str='k-means++'):
    """
    Plot the elbow curve & compute silhouette scores to find optimal 'k' for k-means
    for a range of k values.

    Parameters:
    -----------
    data : pd.DataFrame
        Dataframe containing processed data on which the k-means model is to be fit.

    k_range : range, default value is range(2, 50)
        Range of 'k' values within which the model fit needs to be evaluated.
```

```
    init : str, default value is 'k-means++'
        {'k-means++', 'random'}, Method for initialisation.

    Returns:
    --------
    knee_elbow.knee : int
        Elbow curve knee point.

    knee_ss.knee : int
        Silhouette score knee point.
    """
    WCSS = []
    SS = []
    for k in k_range:
        km = KMeans(n_clusters = k, init=init, max_iter=20000, n_init=100, random_state=1500)
        km.fit(data)
        WCSS += [km.inertia_]
        SS += [silhouette_score(data, labels=km.labels_, random_state=1500)]
    # plot the charts
    fig1 = plt.figure(figsize=(18, 8))
    fig1.subplots_adjust(hspace=.5, wspace=.25)
    plt.clf()
    # Elbow curve for WCSS scores
    plt.subplot(1, 2, 1)
    plt.plot(k_range, WCSS, color='violet', marker='*')
    plt.xlabel('k', weight='bold', fontsize=10)
    plt.ylabel('Within Cluster Sum of Squares', weight='bold', fontsize=10)
    plt.title(f'Elbow Curve: Plot of k vs WCSS (init={init})\n',
    weight='bold', fontsize=12)
    knee_elbow = KneeLocator(k_range, WCSS, S=10, online=True,
    curve='convex', direction='decreasing', interp_method='interp1d')
    if knee_elbow.knee!=None and knee_elbow.knee in range(len(WCSS)):
        plt.axvline(x=knee_elbow.knee, ymax=0.95, ymin=0.05, color='brown', ls='--')
        print(f'Knee point, WCSS : {knee_elbow.knee, WCSS[knee_elbow.knee]}')
        plt.figtext(0.25, 0.25, f'optimal k = {knee_elbow.knee}',
        bbox=dict(facecolor='red', alpha=0.3), fontsize=12)

    # Silhouette scores plot
    plt.subplot(1, 2, 2)
    plt.plot(k_range, SS, color='darkgreen', marker='o')
    plt.xlabel('k', weight='bold', fontsize=10)
    plt.ylabel('Silhouette score', weight='bold', fontsize=10)
    plt.title(f'Silhouette score analysis for optimal k (init={init})\n',
    weight='bold', fontsize=12)
    knee_ss = KneeLocator(k_range, SS, S=10, online=True, curve='concave',
    direction='increasing', interp_method='interp1d')
    if knee_ss.knee!=None and knee_ss.knee in range(len(SS)):
        print(f'Knee point, SS : {knee_ss.knee, SS[knee_ss.knee]}')
        plt.figtext(0.75, 0.25, f'optimal k = {knee_ss.knee}',
        bbox=dict(facecolor='red', alpha=0.3), fontsize=12)
    fig1.tight_layout(pad=2.0)
    plt.show()
    return knee_elbow.knee, knee_ss.knee

def km_final_fit(data_df: pd.DataFrame, opt_k: int):
    """
    Fit k-means model on the processed data and
```

```
    plot all charts for the model cluster predictions

    Parameters:
    -----------
    data : pd.DataFrame
        Dataframe containing processed data on which the k-means model is to be fit.

    opt_k : int
        Optimal 'k' value computed by find_optimal_k().

    Returns:
    --------
    km_final_clusters_df : pd.DataFrame
        DF with cluster labels.
    """
    # Fit the K-means model on the data with the optimal 'k'
    k_final = opt_k # we get this value from either Elbow test or Silhouette score analysis
    km_final = KMeans(n_clusters = k_final, init='k-means++',
    max_iter=20000, n_init=100, random_state=1500)
    km_final = km_final.fit(data_df)
    km_final_clusters_df = pd.DataFrame(index=data_df.index, columns=['km_cluster'])
    km_final_clusters_df['km_cluster'] = km_final.labels_

    # TSNE
    print('\n-------------------------------------\n')
    print('TSNE plot for the model')
    labels_df = plot_TSNE(data_df, km_final.labels_)
    # plot cluster count (bar chart)
    print('\n-------------------------------------\n')
    print('Cluster counts bar chart')
    plot_cluster_counts(labels_df)
    # plot cluster members
    print('\n-------------------------------------\n')
    print('Cluster member price movements: sample plots of 4 smallest clusters')
    plot_cluster_members(labels_df)
    return km_final_clusters_df

def find_and_fit_optimal_kmeans(final_features_df: pd.DataFrame, k_range=range(2, 25),
init='k-means++'):
    print('-' * 50)
    print('Find the optimal k value for fitting K-means model\n')
    opt_k_elbow, opt_k_SS = find_optimal_k(final_features_df, k_range=k_range, init=init)
    print(f'Optimal k based on Elbow curve: {opt_k_elbow}\n')
    print(f'Optimal k based on Silhouette score analysis: {opt_k_SS}\n')

    km_cluster_df = None
    if opt_k_SS is not None or opt_k_elbow is not None:
        k_final = 0
        if opt_k_elbow is None:
            k_final = opt_k_SS
        elif opt_k_SS is None:
            k_final = opt_k_elbow
        else:
            k_final = min(opt_k_elbow, opt_k_SS)
        # fit K-means using the derived optimal k value
        km_cluster_df = km_final_fit(final_features_df, opt_k=k_final)
    else:
        raise Exception("No optimal value for k found!")
```

```
    return km_cluster_df

def optics_fit(data_df: pd.DataFrame, max_eps: float=np.inf, min_samples: int=5):
    """
    Fit OPTICS clustering algorithm over the data;
    Plot all charts for the model predictions

    Parameters:
    -----------
    data_df : pd.DataFrame
        Dataframe containing processed data on which the density based OPTICS model is to be fit.

    max_eps : int
        The maximum distance between two samples for one to be considered
        as in the neighborhood of the other.
        Default value of np.inf will identify clusters across all scales;
        reducing max_eps will result in shorter run times.

    min_samples : int, default is 5
        The number of samples in a neighborhood for a point to be considered as a core point.
        Also, up and down steep regions can't have more than
        min_samples consecutive non-steep points.
        Expressed as an absolute number or a fraction of the number of samples
        (rounded to be at least 2).

    Returns:
    --------
    db_final_clusters_df : pd.DataFrame
        DF with computed cluster labels.
    """
    print('Running density based clustering - OPTICS model')
    print('--'*50)
    print(f'max_eps: {max_eps}\nmin_samples: {min_samples}')
    db = OPTICS(max_eps=max_eps, min_samples=min_samples, xi=0.04).fit(data_df)
    labels = db.labels_
    print(f'model fit params: {db.get_params()}\n')
    # Compute Silouette score
    sil_score = silhouette_score(data_df, labels=labels, random_state=1500)
    print(f'\nThe Silhouette Score for our OPTICS model fit on preprocessed & scaled data :
    f'{sil_score}')
    # Compute Calinski Harabasz score
    cal_hb_score = calinski_harabasz_score(data_df, labels=labels)
    print(f'\nThe Calinski Harabasz Score for our OPTICS model fit on '
    f'preprocessed & scaled data : {cal_hb_score}')

    # TSNE
    print('--'*100)
    print('TSNE plot for the model')
    labels_df = plot_TSNE(data_df, labels)
    # plot cluster count (bar chart)
    print('--'*100)
    print('Cluster counts bar chart')
    plot_cluster_counts(labels_df)
    # plot cluster members
    print('--'*100)
    print('Cluster member price movements: sample plots of 4 least dense clusters')
    plot_cluster_members(labels_df)
```

```
    db_final_clusters_df = pd.DataFrame(index=data_df.index, columns=['db_cluster'])
    db_final_clusters_df['db_cluster'] = labels
    return db_final_clusters_df

def agg_hc_fit(data_df: pd.DataFrame, n_clusters: int=10, linkage: str='average'):
    """
    Fit Agglomerative Clustering algorithm over the data;
    Plot all charts for the model predictions

    Parameters:
    -----------
    data_df : pd.DataFrame
        Dataframe containing processed data on which Hierarchical Agglomerative Clustering
        model is to be fit.

    n_clusters : int, default is 10
        The number of clusters to find.

    linkage : str, default is 'average'
        {'ward', 'complete', 'average', 'single'},
        Which linkage criterion to use. The linkage criterion determines which
        distance to use between sets of observation.
        The algorithm will merge the pairs of cluster that minimize this criterion.

            1. 'ward' minimizes the variance of the clusters being merged.

            2. 'average' uses the average of the distances of each observation of the two sets.

            3. 'complete' or 'maximum' linkage uses the maximum distances between all
            observations of the two sets.

            4. 'single' uses the minimum of the distances between all observations of the two sets.

    Returns:
    --------
    hc_final_clusters_df : pd.DataFrame
        DF with computed cluster labels.
    """
    print('Running hierarchical clustering (bottom-up approach) - Agglomerative model')
    print('--'*50)
    print(f'n_clusters: {n_clusters}')
    hc = AgglomerativeClustering(n_clusters=n_clusters, metric='euclidean', linkage=linkage).fit(
    data_df)
    labels = hc.labels_
    print(f'model fit params: {hc.get_params()}\n')
    # Compute Silouette score
    sil_score = silhouette_score(data_df, labels=labels, random_state=1500)
    print(f'\nThe Silhouette Score for our Agglomerative model fit on preprocessed &'
    f'scaled data : {sil_score}')
    # Compute Calinski Harabasz score
    cal_hb_score = calinski_harabasz_score(data_df, labels=labels)
    print(f'\nThe Calinski Harabasz Score for our Agglomerative model fit on preprocessed & '
    f'scaled data : {cal_hb_score}')

    # TSNE
    print('--'*100)
    print('TSNE plot for the model')
```

```
    labels_df = plot_TSNE(data_df, labels)
    # plot cluster count (bar chart)
    print('--'*100)
    print('Cluster counts bar chart')
    plot_cluster_counts(labels_df)
    # plot cluster members
    print('--'*100)
    print('Cluster member price movements: sample plots of 4 least dense clusters')
    plot_cluster_members(labels_df)

    hc_final_clusters_df = pd.DataFrame(index=data_df.index, columns=['hc_cluster'])
    hc_final_clusters_df['hc_cluster'] = labels
    return hc_final_clusters_df

def compare_clustering_results(km_cluster_df: pd.DataFrame, db_cluster_df: pd.DataFrame,
hc_cluster_df: pd.DataFrame, sp500_stocks_profile_df: pd.DataFrame):
    all_clustering_df = pd.concat([km_cluster_df, db_cluster_df, hc_cluster_df], axis=1)
    all_clustering_df['sector'] = [sp500_stocks_profile_df[sp500_stocks_profile_df['ticker'] == tkr]
    all_clustering_df = all_clustering_df[['sector', 'km_cluster', 'db_cluster', 'hc_cluster']]

    print("Head of combined clustering dataframe:")
    print(all_clustering_df.head(15))
    print("\nSector-wise breakdown for k-means clusters:")
    print(pd.crosstab(all_clustering_df.sector,
    all_clustering_df.km_cluster).style.highlight_max(color='orange', axis=0))

    print("\nSector-wise breakdown for OPTICS clusters:")
    print(pd.crosstab(all_clustering_df.sector,
    all_clustering_df.db_cluster).style.highlight_max(color='green', axis=0))

    print("\nSector-wise breakdown for Agglo clusters:")
    print(pd.crosstab(all_clustering_df.sector,
    all_clustering_df.hc_cluster).style.highlight_max(color='indigo', axis=0))

def get_final_clabel_profile_df(selected_label_df: pd.DataFrame, target_ratios: list):
    """
    Returns a final cluster profile dataframe containing the chosen cluster labels,
    mean ratios data, sector and company info.
    Mainly intended for EDA and drawing insights.

    Parameters:
    -----------
    selected_label_df : pd.DataFrame
        Cluster label DF for a selected clustering method.

    target_ratios: list
        Selecting the relevant ratios.

    Returns:
    --------
    final_cluster_df : pd.DataFrame
        DF with computed cluster labels and stock profile details appended to it.
    """
    conn = sqlite3.connect('./data/pairs_trading.db')
    sp500_stocks_profile_df = pd.read_sql_query("SELECT * FROM stocks_profile", conn)
    stock_ratios_df = pd.read_sql_query("SELECT * FROM stocks_hist_ratios", conn)

    conn.close()
```

```
    final_cluster_df = pd.DataFrame(index=selected_label_df.index, columns=['company_name', 'sector'
    final_cluster_df['company_name'] = [sp500_stocks_profile_df[sp500_stocks_profile_df.ticker
    == tkr]['company_name'].values[0] for tkr in final_cluster_df.index]
    final_cluster_df['sector'] = [sp500_stocks_profile_df[sp500_stocks_profile_df.ticker == tkr]
    ['sector'].values[0] for tkr in final_cluster_df.index]
    final_cluster_df['cluster'] = [selected_label_df.loc[tkr][0] for tkr in final_cluster_df.index]
    final_cluster_df['bus_desc'] = [sp500_stocks_profile_df[sp500_stocks_profile_df.ticker == tkr]
    ['business_desc'].values[0] for tkr in final_cluster_df.index]
    # fetch the mean target ratio measures for all stocks
    temp_rat = stock_ratios_df[['ticker', 'Fiscal Date Ending'] + target_ratios].iloc[:,
    2:].apply(pd.to_numeric, errors='coerce', downcast='float')
    temp_rat = temp_rat.fillna(0)
    temp_rat = pd.concat([stock_ratios_df[['ticker', 'Fiscal Date Ending']], temp_rat], axis=1)
    temp_rat = temp_rat[temp_rat['Fiscal Date Ending'] < 2020]
    temp_rat = temp_rat.reset_index(drop=True)
    ratios_10y_mean_df = temp_rat.drop(columns=['Fiscal Date Ending']).groupby(by=['ticker']).mean()

    final_cluster_df = pd.concat([final_cluster_df, ratios_10y_mean_df], axis=1, join='inner')

    print("\nSector-wise distribution among the OPTICS clusters:")
    print(pd.crosstab(final_cluster_df.sector,
    final_cluster_df.cluster).style.highlight_max(color='darkgreen', axis=0))
    return final_cluster_df

def boxplot_cluster_fin_ratios(data: pd.DataFrame, target_ratios: list):
    """
    Plot the boxplots of all target final ratios for each cluster.

    Parameters:
    -----------
    data : pd.DataFrame
        DF with computed cluster labels and stock profile details appended to it.
        This is output by get_final_clabel_profile_df()

    target_ratios : list
        List of target ratios.
    """
    plt.figure(figsize=(18, 24))
    plt.subplots_adjust(hspace=.5, wspace=.25)
    plt.clf()
    counter = 0
    for row in range(5):
        for col in range(2):
            if counter <len(data.cluster.value_counts()):
                plt.subplot(5, 2, counter+1)
                sns.boxplot(y=data[target_ratios[counter]], x=data.cluster)
            plt.title(f'Box plot of {target_ratios[counter]}\n', weight='bold', fontsize=14)
            plt.xlabel('Clusters', weight='bold', fontsize=12)
            plt.ylabel(f'{target_ratios[counter]}', weight='bold', fontsize=12)
            counter += 1
    plt.tight_layout(pad=2.0)
    plt.show()

def plot_cluster_wordclouds(data_df: pd.DataFrame):
    """
    Plot wordclouds of business descriptions of companies in each cluster

    Parameters:
```

```
    -----------
    data_df : pd.DataFrame
        DF with computed cluster labels and stock profile details appended to it.
        This is output by get_final_clabel_profile_df()
    """
    # Plot wordclouds within each cluster
    print('-'*50)
    print('Plot wordclouds of business descriptions of companies in each cluster\n')

    fig1 = plt.figure(figsize=(18, 20))
    fig1.subplots_adjust(hspace=.25, wspace=.05)
    plt.axis('off')
    plt.clf()
    cluster_labels = data_df.cluster.value_counts().index
    cluster_labels = [label for label in cluster_labels if label != -1]
    for i in range(len(cluster_labels)):
        fig1.add_subplot(5, 3, i+1)
        wc = WordCloud(background_color='black', width=700, height=400, max_words=400,
        min_font_size=16, max_font_size=50, random_state=1500)
        full_text = '; '.join(data_df[data_df['cluster'] == cluster_labels[i]]['bus_desc'])
        clean_text = text_cleaning(full_text)
        wc = wc.generate(str(clean_text))
        plt.axis('off')
        plt.imshow(wc, cmap=cm.Dark2_r)
        plt.title(f'Top words in cluster {cluster_labels[i]}', weight='bold', fontsize=16)
        i += 1
    plt.tight_layout(pad=2.0)
    plt.show()


def filter_stock_profiles(sp500_comp_profile_df: pd.DataFrame):
    # Filter out only the necessary information
    filtered_stock_profiles = sp500_comp_profile_df[['ticker', 'company_name', 'sector',
    'sub_industry', 'business_desc']]

    # Check for any NaN values
    print(f'Total count of Nan values in stock profile data: {filtered_stock_profiles.isna().sum().s
    print(filtered_stock_profiles.isna().sum())

    # Drop tickers with missing values in business_desc column only
    filtered_stock_profiles.dropna(axis=0, subset=['business_desc'], inplace=True)
    print(f'Total stocks in consideration after dropping the ones with '
    f'missing business descriptions: {filtered_stock_profiles.shape[0]}')

    return filtered_stock_profiles


def preprocess_text(text, flg_stemm=False, flg_lemm=True):
    """
    Clean & preprocess input string.
    Note:   You should not set both flg_stemm & flg_lemm to be True.
            Only one of them can be True at a time.
    """
    # clean (convert to lowercase and remove punctuations and special characters and then strip)
    text = re.sub(r'[^\w\s]', '', str(text).lower().strip())

    # Tokenize (convert from string to list)
    tokens = text.split()
```

```
    # remove Stopwords
    # adding more words to the default list
    stop_words.extend(['founded', 'firm', 'company', 'llc', 'inc', 'incorporated',
                       'multinational', 'corporation', 'commonly', 'headquartered'])
    if stop_words is not None:
        tokens = [word for word in tokens if word not in stop_words]

    # Stemming (remove -ing, -ly, ...)
    if flg_stemm == True:
        ps = nltk.stem.porter.PorterStemmer()
        tokens = [ps.stem(word) for word in tokens]

    # Lemmatisation (convert the word into root word)
    if flg_lemm == True:
        lem = nltk.stem.wordnet.WordNetLemmatizer()
        tokens = [lem.lemmatize(word) for word in tokens]

    # back to string from list
    text = ' '.join(tokens)
    return text


def apply_preprocessing(stock_profiles: pd.DataFrame):
    stock_profiles['bd_clean'] = stock_profiles['business_desc'].apply(lambda x: preprocess_text(x))
    return stock_profiles


def count_unique_words(stock_profiles: pd.DataFrame):
    count = len(set([ele for arr in list(stock_profiles["bd_clean"].apply(
    lambda x: str(x).split(" "))) for ele in arr]))
    print(f'Total count of unique words (unigrams) found in our corpus of business '
    f'descriptions across all stocks: {count}')
    return count


def tfidf_transform(stock_profiles: pd.DataFrame, min_df=1, max_df=0.1, ngram_range=(2, 4)):
    tfidf = feature_extraction.text.TfidfVectorizer(stop_words='english', max_features=10000,
    min_df=min_df, max_df=max_df,strip_accents='unicode', ngram_range=ngram_range)
    tfidf_sparse = tfidf.fit_transform(stock_profiles['bd_clean'])
    return tfidf_sparse


def visualize_sparse_matrix(tfidf_sparse):
    sns.heatmap(tfidf_sparse.todense()[:, np.random.randint(0, tfidf_sparse.shape[1], 10)] == 0, vmi
        .set_title('Sparse Matrix Sample')
```

```
#PATH: utils/pairs_trading.py
##--This script contains code for pairs trading strategy--

import pandas as pd
import numpy as np
import datetime as dt
pd.set_option('display.max_rows', 2000)
pd.set_option('display.max_columns', 1000)
pd.set_option('display.width', 1000)

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
import sqlite3
sns.set(style="white", rc={"figure.figsize":(8, 4)})
plt.style.use('ggplot') # fivethirtyeight, ggplot, dark_background, classic,


from statistics import mean
from statsmodels.regression.linear_model import OLS
from statsmodels.tsa.stattools import coint
import math

conn = sqlite3.connect('./data/pairs_trading.db')

stocks_hist_price_df = pd.read_sql_query("SELECT * FROM stocks_hist_price", conn,
parse_dates=['date'])
stocks_hist_price_df.date = pd.to_datetime(stocks_hist_price_df.date, format='ISO8601')

# fetch full historical adj close price
full_hist_close_df = stocks_hist_price_df.pivot_table(values='Adj Close',
index='date', columns='ticker')
full_hist_close_df.index = full_hist_close_df.index.date
full_hist_close_df.interpolate(method='linear', axis=1, inplace=True)

cluster_df = pd.read_csv('./data/db_clabel_10y_data.csv')
target_cluster_df = cluster_df[cluster_df.cluster!=-1].reset_index(drop=True)

"Pair selection method"
"select a pair with lowest p-value from each cluster"
def cluster_pair_selection(cluster_df: pd.DataFrame, significance=0.05,
                    start_day=dt.datetime(2010, 1, 1).date(),
                    end_day=dt.datetime(2019, 12, 31).date()):
    final_stock_pairs = []
    cluster_labels = cluster_df.cluster.value_counts().index.to_list()
    temp_price_df = full_hist_close_df.loc[start_day: end_day]

    for i in range(len(cluster_labels)):
        # loop over clusters to get best coint pair in each cluster
        tickers = target_cluster_df[target_cluster_df.cluster ==
        cluster_labels[i]]['ticker'].values
        p_values = []
        coint_pairs = []
        for m in range(len(tickers)):
            for n in range(m+1,len(tickers)):
                p_val = coint(temp_price_df[tickers[m]], temp_price_df[tickers[n]])[1]
                p_values.append(p_val)
                coint_pairs.append([tickers[m], tickers[n]])
```

```
        if len(coint_pairs) > 0:
            if np.min(p_values) < significance:
                index = np.where(p_values == np.min(p_values))[0][0]
                final_stock_pairs.append(coint_pairs[index])

    return final_stock_pairs

"Calculate benchmark return and plot the relevant performance charts"
def plot_benchmark_ret(conn):
    spy_index_df = pd.read_sql_query("SELECT * FROM SPY_hist_price", conn, parse_dates=['date'])
    spy_index_df.date = pd.to_datetime(spy_index_df.date, format='ISO8601')
    spy_index_df.date = spy_index_df.date.apply(lambda x: x.date())
    temp_df = spy_index_df[(spy_index_df.date > dt.datetime(2020, 1, 1).date()) &
    (spy_index_df.date < dt.datetime(2021, 1, 1).date())]
    temp_df['cum_ret'] = temp_df['simple_return'].cumsum()
    spy_sharpe = np.nanmean(temp_df['simple_return'])/np.nanstd(temp_df['simple_return'])

    # plot benchmark
    plt.title(f"Benchmark return for SPY tracking S&P 500 index\n",
                fontweight='bold', fontsize=12)
    plt.plot(temp_df['date'], temp_df['simple_return'] * 100, label='Daily returns')
    plt.plot(temp_df['date'], temp_df['cum_ret'] * 100, linestyle = '--',
    label='Cummulative returns')
    plt.ylabel("Percentage return (%)")
    plt.figtext(0.65, 0.4, f'Sharpe ratio = {spy_sharpe:.4f}',
    bbox=dict(facecolor='orange', alpha=0.3), fontsize=10)
    plt.legend(loc='best')
    plt.show()


class Pair:
    P_VALUE_THRESHOLD = 0.05 # significance level
    MIN_MEAN_CROSSES = 12 # minimum number of mean crosses in the signal
    TRADING_BOUND = 1 # STD level which signifies the trading bounds
    EXIT_PROFIT = 0 # level at which to exit and book profits (close the position)
    STOP_LOSS = 2 # STD level which signifies the stoploss bounds
    RETURN_ON_BOTH_LEGS = False # method of computing returns (should both long & short legs
    be considered)
    WAITING_DAYS = 15 # wait time period for a signal hold its state after crossing a level
    before which to close the trade
    TRADING_START_DATE = dt.datetime(2020, 1, 1).date() # backtest period
    TRADING_END_DATE = dt.datetime(2021, 1, 1).date()
    TC = 1 # transaction costs (defined as a percentage of capital invested)

    def __init__(self,
                 stockX,
                 stockY,
                 trading_start_date = TRADING_START_DATE,
                 trading_end_date = TRADING_END_DATE,
                 waiting_days = WAITING_DAYS,
                 trading_bound = TRADING_BOUND,
                 exit_profit = EXIT_PROFIT,
                 stop_loss = STOP_LOSS,
                 return_on_both_legs = RETURN_ON_BOTH_LEGS,
                 tc = TC):
        # constructor
        self.name = f"{stockX.name}, {stockY.name}"
```

```
        self.stockX_trading = stockX[trading_start_date : trading_end_date]
        self.stockY_trading = stockY[trading_start_date : trading_end_date]
        self.waiting_days = waiting_days
        self.trading_start_date = trading_start_date
        self.trading_end_date = trading_end_date
        self.exit_profit = exit_profit
        self.trading_bound = trading_bound
        self.stop_loss = stop_loss
        self.return_on_both_legs = return_on_both_legs
        self.tc = tc
        self.error = False

        try:
            beta = OLS(stockY[trading_start_date : trading_end_date],
            stockX[trading_start_date : trading_end_date]).fit().params[0]
            self.spread = stockY[trading_start_date : trading_end_date] - beta *
            stockX[trading_start_date : trading_end_date]
            self.normalized_spread_trading = (self.spread - self.spread.mean()) / self.spread.std()
            self.p_value = coint(stockX, stockY)[1]
            self.generate_trading_signals()
        except Exception as e:
            print(e)
            print(f"Error encountered with pair "
            f"[{self.stockX_trading.name}, {self.stockY_trading.name}]")
            self.error = True

    def eligible(self, p_value_threshold = P_VALUE_THRESHOLD, min_mean_crosses = MIN_MEAN_CROSSES):
        # Check for the pair eligibility (if its p_val < threshold & if it meets
        the min mean crosses threshold)
        if self.error:
            return False
        elif self.p_value <= p_value_threshold and self.mean_crosses >= min_mean_crosses:
            return True
        return False

    def __level_crosses(self, series, level = 2):
        # Identify & record the changes in the signal movements as and when it crosses
        # either above or below a defined level
        change = []
        for i, el in enumerate(series):
            if i != 0 and el > level and series[i-1] < level:
                # if signal crosses above the level (going upwards), record it as '1'
                change.append(1)
            elif i != 0 and el < level and series[i-1] > level:
                # if signal crosses below the level (going downwards), record it as '-1'
                change.append(-1)
            else:
                # else, record it as no change, '0'
                change.append(0)
        return change

    def __average_holding_period(self):
        # compute the average holding period for the pair,
        # based on their opening and closing positions recorded
        holding_periods = []
        for closed_date in self.closed_positions:
            open_date = list(filter(lambda x: x < closed_date, self.open_positions))[-1]
            holding_periods.append(closed_date - open_date)
```

```
        return np.mean(np.array(holding_periods))

    def __calculate_returns(self, pos_y, pos_x, i):
        # function to compute the total returns for the pair

        # first, calculate PnL for the pair either when a stoploss is recorded,
        # or when the position is closed with profit booking
        if pos_x[2] == 'l':
            # long on X (meaning short Y - short the spread)
            # cost of long = opening price * number of stocks bought
            cost_long = pos_x[0] * pos_x[1]
            # cost of short => opening price * number of stocks
            #(this is a simplified consideration)
            cost_short = pos_y[0] * pos_y[1]
            profit = (self.stockX_trading[i] - self.pos_x[0]) * pos_x[1] + (self.pos_y[0]
            - self.stockY_trading[i]) * pos_y[1]
        else:
            # long on Y (meaning long spread - short X)
            cost_long = pos_y[0] * pos_y[1]
            cost_short = pos_x[0] * pos_x[1]
            profit = (self.pos_x[0] - self.stockX_trading[i]) * pos_x[1] +
            (self.stockY_trading[i] - self.pos_y[0]) * pos_y[1]

        # second, compute returns considering the transaction costs as well
        if self.return_on_both_legs:
            # (computes the unlevered rets)
            # computing total rets considering both the legs of the trade
            # (long & short) to compute initiation costs
            return_before_tc = (profit / (cost_long + cost_short)) * 100
            return return_before_tc * (1.0 - (self.tc / 100))
        else:
            # calculate the return only on the cost of long position (computes the levered rets)
            return_before_tc = (profit / cost_long) * 100
            return return_before_tc * (1.0 - (self.tc / 100))

    def generate_trading_signals(self):
        # generate the trading signals for the pair (basically computes the 1, -1 & 0
        #values for the trade signal based on level crossing)
        self.upper_trading = self.__level_crosses(self.normalized_spread_trading, level =
        self.trading_bound)
        self.lower_trading = self.__level_crosses(self.normalized_spread_trading, level = -
        self.trading_bound)
        self.upper_stop = self.__level_crosses(self.normalized_spread_trading, level = self.stop_los
        self.lower_stop = self.__level_crosses(self.normalized_spread_trading, level = -
        self.stop_loss)
        self.mean = self.__level_crosses(self.normalized_spread_trading, level = self.exit_profit)

        # record the count of mean crosses
        self.mean_crosses = self.mean.count(1) + self.mean.count(-1)

        open_position = False # flag to check for position status
        # entry_level = 0 # pos opening entry level

        # placeholders
        self.stop_losses = []
        self.open_positions = []
        self.closed_positions = []
        returns = []
```

```
i = 0
while i < len(self.normalized_spread_trading):
    # loop over every data point in the normalised spread signal (each date until the end)
    if open_position:
        # if in an open position,
        # 1. if signal crosses above upper stoploss or crosses below lower stoploss bounds
        #- book Loss & record rets, update stoplosses & wait for reversal
        # 2. if signal crosses at the mean bounds - close pos & book Profit & record rets
        # 3. else - nothing, so record no change with '0' rets
        if self.upper_stop[i] == 1 or self.lower_stop[i] == -1:
            # STOP LOSS (CLOSE WITH LOSS & WAIT FOR REVERSAL)
            open_position = False
            returns.append(self.__calculate_returns(pos_y= self.pos_y, pos_x=self.pos_x, i=i
            self.stop_losses.append(i)
            # wait for signal to reverse its course and get back within the bounds
            i += self.waiting_days
            returns.extend([0] * (self.waiting_days - 1))
            continue
        elif self.mean[i] != 0:
            # CLOSING WITH PROFIT
            open_position = False
            returns.append(self.__calculate_returns(pos_y= self.pos_y, pos_x=self.pos_x, i=i
            self.closed_positions.append(i)
        else:
            returns.append(0)
    else:
        # if not in an open position,
        # enter into a pos if signal crosses below upper trading bound - short the spread
        # if signal above lower trading bound - go long the spread
        # else, no change,
        # record returns as '0' (since in any case we are just opening a fresh positiion
        #& not closing to be able to compute rets)
        if self.upper_trading[i] == -1 or self.lower_trading[i] == 1:
            #ENTERING THE POSITION
            open_position = True
            self.open_positions.append(i)
            #the return will not depend on the amount we fixed here
            approx_amount = 100000
            open_price_x = self.stockX_trading[i]
            open_price_y = self.stockY_trading[i]
            b = open_price_y / open_price_x
            a = approx_amount / (open_price_y + b * open_price_x)
            # we assume that you cannot buy portion of stocks, and the fixed
            # amount is approximated (<1% error assumption!)
            number_stocks_y = math.ceil(a)
            number_stocks_x = math.ceil(a*b)
            total_eff_amount = number_stocks_y * open_price_y + number_stocks_x *
            open_price_x
            # placeholders for recording trade details (price & shares) when a
            #position was opened; 'l' for long and 's' for short
            if self.upper_trading[i] == -1:
                #LONG X, SHORT Y
                self.pos_x = (open_price_x, number_stocks_x, 'l')
                self.pos_y = (open_price_y, number_stocks_y, 's')
            elif self.lower_trading[i] == 1:
                #SHORT X, LONG Y
                self.pos_x = (open_price_x, number_stocks_x, 's')
```

```
                    self.pos_y = (open_price_y, number_stocks_y, 'l')
                # record the spread level for entering into position
                # entry_level = self.spread[i]
            # here in this step, we just opened a fresh pos, so record rets as '0'
            returns.append(0)
        # step up to the next date
        i += 1

    self.profitable_trades_perc = len(self.closed_positions) /
    (len(self.closed_positions) + len(self.stop_losses)) * 100
    self.average_holding_period = self.__average_holding_period()

    self.returns_series = pd.Series(index = self.stockX_trading.index, data = returns)
    self.cum_returns = self.returns_series.cumsum()

def plot_pair(self):
    # method to plot the price movement charts for the pair
    fig, (ax_stockX, ax_spread) = plt.subplots(2, 1)
    fig.suptitle(t=f'Price movement and trading signal charts for the pair
    {self.stockX_trading.name, self.stockY_trading.name}',
                 fontsize=16, fontweight='bold')

    ax_stockX.title.set_text("Stocks prices")
    ax_spread.title.set_text("Normalized spread")

    plt_X = ax_stockX.plot(self.stockX_trading, color = "b", label = self.stockX_trading.name)
    ax_stockY = ax_stockX.twinx()
    plt_Y = ax_stockY.plot(self.stockY_trading, color = "y", label = self.stockY_trading.name)
    # Solution for having two legends
    leg = plt_X + plt_Y
    labs = [l.get_label() for l in leg]
    ax_stockX.legend(leg, labs, loc=0)


    ax_spread.plot(self.normalized_spread_trading[self.trading_start_date :
    self.trading_end_date])
    ax_spread.axhline(self.trading_bound, linestyle = '--', color = "g",
    label = "Trading bound")
    ax_spread.axhline(-self.trading_bound, linestyle = '--', color = "g")
    ax_spread.axhline(self.stop_loss, linestyle = '--', color = "r", label = "Stop loss")
    ax_spread.axhline(-self.stop_loss, linestyle = '--', color = "r")
    ax_spread.plot_date([self.normalized_spread_trading.index[i] for i in self.open_positions],
    marker = '^', markeredgecolor = 'b', markerfacecolor = 'b', markersize = 16)
    ax_spread.plot_date([self.normalized_spread_trading.index[i] for i in self.closed_positions]
    [self.normalized_spread_trading[i] for i in self.closed_positions], label = 'Closed
    position', marker = 'P', markeredgecolor = 'g', markerfacecolor = 'g', markersize = 16)
    ax_spread.plot_date([self.normalized_spread_trading.index[i] for i in self.stop_losses],
    [self.normalized_spread_trading[i] for i in self.stop_losses],
    label = 'Stop loss', marker =
    'X', markeredgecolor = 'r', markerfacecolor = 'r', markersize = 16)
    ax_spread.legend(loc=0)

    fig.set_size_inches(16, 10, forward = True)
    fig.show()
    plt.tight_layout()

def __repr__(self):
    s = f"Pair [{self.stockX_trading.name}, {self.stockY_trading.name}]"
```

```
        s += f"\n\tp-value: {self.p_value}"
        s += f"\n\tMean crosses: {self.mean_crosses}"
        s += f"\n\tPair eligible: {self.eligible()}"
        s += f"\n\tProfitable trades (%): {self.profitable_trades_perc}"
        s += f"\n\tAverage holding period (days): {self.average_holding_period}"
        return s


class Portfolio:
    def __init__(self, stocks_df: pd.DataFrame, pairs_list: list):
        if not isinstance(stocks_df, pd.core.frame.DataFrame):
            raise Exception("Symbols must be provided in a Pandas DataFrame")

        self.time_series_df = stocks_df
        # self.time_series_df.dropna(inplace = True)
        self.symbols_list = self.time_series_df.columns
        # self.all_possible_pairs = list(combinations(self.symbols_list, 2))
        self.defined_pairs = pairs_list
        self.selected_pairs = list()

        for i, pair_symbols in enumerate(self.defined_pairs):
            print(f"{i}/{len(self.defined_pairs)}")
            pair = Pair(self.time_series_df[pair_symbols[0]],
                        self.time_series_df[pair_symbols[1]])
            if pair.eligible():
                self.selected_pairs.append(pair)

        self.calculate_portfolio_return()

    def calculate_portfolio_return(self):
        # compute portfolio return & cumm return  metrics
        data = dict()
        for pair in self.selected_pairs:
            data[pair.name] = pair.returns_series
        df_return = pd.DataFrame(data = data)
        df_return['Return'] = df_return.mean(axis = 1)
        self.returns = df_return['Return']
        df_return['Cumulative Return'] = df_return['Return'].cumsum()
        self.cum_return = df_return['Cumulative Return']
        self.sharpe_ratio = np.nanmean(self.returns)/np.nanstd(self.returns)

    def plot_portfolio(self):
        # plot portfolio
        plt.title(f"Return of a selective portfolio of ({len(self.selected_pairs)}) "
        f"pairs of cointegrated stocks\n", fontweight='bold', fontsize=12)"
        plt.plot(self.returns, label='Daily returns')
        plt.plot(self.cum_return, linestyle = '--', label='Cummulative returns')
        plt.ylabel("Percentage return (%)")
        plt.legend(loc='best')
        plt.figtext(0.65, 0.4, f'Sharpe ratio = {self.sharpe_ratio:.4f}',
        bbox=dict(facecolor='orange', alpha=0.3), fontsize=10)
        plt.show()

    def plot_pairs(self):
        # plot pairs price movements
        for pair in self.selected_pairs:
            pair.plot_pair()
conn.close()
```