# Machine Learning-Based Pair Trading Strategy

Yichen Liu

January 23, 2024

**Abstract**

In this article, we apply four unsupervised learning clustering methods to categorize SP500 stocks and pinpoint suitable trading pairs within these groups. We follow crucial steps, including data cleaning, dividing it into training and backtesting sets(to avoid cheating!), selecting optimal parameters or clusters, training models, and choosing the best one to achieve the clustering result. Ultimately, we introduce two pair trading strategies and assess their performance by backtesting the selected pairs. This approach aims to enhance the transparency of the trading pair selection process while evaluating the effectiveness of applying unsupervised learning to pair trading.

## 1 Introduction to Pair Trading

Pairs Trading, established in the 1980s, stands as a widely recognized investment strategy utilized extensively as a crucial long/short equity investment tool by hedge funds and institutional investors.

This strategy involves two primary steps:

1. First, identifying two securities, such as stocks, whose price series exhibit similar behavior or appear to be correlated. This suggests that both securities are influenced by related risk factors and tend to react similarly.

2. The second step seizes on the idea that if two securities historically move together, any deviation from this pattern presents a trading opportunity. Monitoring the changing spread between the pair's constituents helps identify anomalies. When detected, a market position is initiated and exited upon spread correction.

To enhance the effectiveness of a Pairs Trading strategy, the identification of optimal pairs is crucial. However, this task is complicated by the difficulty in discovering profitable pairs, often due to the close monitoring of such opportunities. The challenge lies in the vast number of potential combinations.

To overcome this obstacle, this paper proposes leveraging Unsupervised Learning to streamline the exploration process. The objective is to employ clustering techniques on securities, irrespective of their sector affiliations, to uncover clusters and subsequently identify lucrative pairs within these groupings.

## 2 Introduction to Unsupervised Learning

### 2.1 Unsupervised Learning

Clustering involves the grouping of entities that share similarities, forming clusters composed of statistically similar data points. Each cluster represents a set of comparable elements. The goal of unsupervised learning is to discover hidden patterns in unlabeled data. Multiple unsupervised learning clustering techniques can be used to complish this.

### 2.2 Example Clustering Algorithms

#### 2.2.1 Agglomerative Hierarchical Clustering

Hierarchical Clustering is a method that groups features into clusters based on their similarity. It can perform the groupage by an agglomerative (bottom-up) or divisive (top-down) approach.

The main advantage that hierarchical clustering has is that it doesn't require us to specify the number of clusters in advance.

Since we want to minimize the variance distance between our clusters we shall go with Ward's linkage:

$$D(C_i, C_j) = \sqrt{\frac{|C_i| \cdot |C_j|}{|C_i| + |C_j|}} \sum_{\mathbf{p} \in C_i \cup C_j} (\mathbf{p} - \mathbf{c})^2$$

Where $\mathbf{c}$ is the centroid of the merged cluster $C_i \cup C_j$.

### 2.2.2 K-means Clustering

K-Means clustering is an algorithm that utilizes unsupervised learning to find and mark K clusters that are specified in advance. K cluster can be found by using either the silhouette or elbow methods. K-means finds clusters that minimize the distances between samples. The algorithem to do K-means can be shown below:

| K-means Clustering Algorithm |
|---|
| 1. For a given cluster assignment C, the total cluster variance is minimized with respect to $m_1$,...,$m_K$ yielding the means of the currently assigned clusters |
| 2. Given a current set of means $m_1$,...,$m_K$, is minimized by assigning each observation to the closest (current) cluster mean. That is, $C(i) = \text{argmin} \, \|x_i - m_k\|_2$ |
| 3. Steps 1 and 2 are iterated until the assignments do not change. |

Table 1: Example Table with K-means Clustering Algorithm

### 2.2.3 DBSCAN

DBSCAN, acronym for Density-Based Spatial Clustering of Applications with Noise, is a clustering algorithm that, when presented with a set of points in a given space, groups together points closely packed. It is a density-based clustering algorithm. The principles are as follows:

1. **Core Point Definition:**

   $N_\varepsilon(p)$ represents the set of data points within the neighborhood of $p$ with a radius of $\varepsilon$.

2. **Core Point Condition:**

   $$\text{If } |N_\varepsilon(p)| \geq \text{MinPts}, \text{ then data point } p \text{ is a core point.}$$

3. **Density-Reachable:**

   If $q \in N_\varepsilon(p)$ and $q$ is a core point, it is said that data point $p$ is density-reachable from $q$.

DBSCAN algorithm partitions data into clusters based on the density-reachable relationship among core points, effectively identifying noise points.

### 2.2.4 Affinity Propagation Clustering

Affinity Propagation Clustering is a clustering algorithm that identifies exemplars in a dataset, assigning each point based on similarity. The algorithm updates responsibilities and availabilities:

1. **Responsibility Update:**

   $$r(i,k) \leftarrow s(i,k) - \max_{k' \neq k} \{a(i,k') + s(i,k')\}$$

2. **Availability Update:**

   $$a(i,k) \leftarrow \min\{0, r(k,k) + \sum_{i' \neq i, i' \neq k} \max\{0, r(i',k)\}\}$$

Here, $r(i,k)$ and $a(i,k)$ determine exemplars and cluster assignments based on pairwise similarities $s(i,k)$.

Affinity Propagation's advantages include automatic determination of cluster number, robustness to outliers, and effectiveness in handling non-flat structures and high-dimensional data.

# 3 How can unsupervised learning solve this problem?

## 3.1 Overview

The main procedure involves data cleaning and splitting, clustering, and identifying trading pairs; subsequently, the selected pairs can be subjected to backtesting.

## 3.2 Data preparation

### 3.2.1 Load Data

First, we load and organize the data to our ideal format.

Listing 1: Load Data

```
path = '/Users/cathyliu/Desktop/NYU-MFE/7773-Machine-Learning/Project-2/'
data = pd.read_csv(path+'all_stocks_5yr.csv')
prices = data[['date','Name','close']]
# Convert 'date' column to datetime type
prices['date'] = pd.to_datetime(prices['date'])
# Set 'date' as the index
prices.set_index('date', inplace=True)
# Pivot the DataFrame
prices = prices.pivot(columns='Name', values='close')
prices.head()
```

| Name | A | AAL | AAP | AAPL | ABBV | ABC | ABT | ACN | ADBE | ADI | ... | XL | XLNX | XOM | XRAY | XRX | XYL | YUM | ZBH | ZION | ZTS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| date | | | | | | | | | | | | | | | | | | | | | |
| 2013-02-08 | 45.08 | 14.75 | 78.90 | 67.8542 | 36.25 | 46.89 | 34.41 | 73.31 | 39.12 | 45.70 | ... | 28.24 | 37.51 | 88.61 | 42.87 | 31.84 | 27.09 | 65.30 | 75.85 | 24.14 | 33.05 |
| 2013-02-11 | 44.60 | 14.46 | 78.39 | 68.5614 | 35.85 | 46.76 | 34.26 | 73.07 | 38.64 | 46.08 | ... | 28.31 | 37.46 | 88.28 | 42.84 | 31.96 | 27.46 | 64.55 | 75.65 | 24.21 | 33.26 |
| 2013-02-12 | 44.62 | 14.27 | 78.60 | 66.8428 | 35.42 | 46.96 | 34.30 | 73.37 | 38.89 | 46.27 | ... | 28.41 | 37.58 | 88.46 | 42.87 | 31.84 | 27.95 | 64.75 | 75.44 | 24.49 | 33.74 |
| 2013-02-13 | 44.75 | 14.66 | 78.97 | 66.7156 | 35.27 | 46.64 | 34.46 | 73.56 | 38.81 | 46.26 | ... | 28.42 | 37.80 | 88.67 | 43.08 | 32.00 | 28.26 | 64.41 | 76.00 | 24.74 | 33.55 |
| 2013-02-14 | 44.58 | 13.99 | 78.84 | 66.6556 | 36.57 | 46.77 | 34.70 | 73.13 | 38.61 | 46.54 | ... | 28.22 | 38.44 | 88.52 | 42.91 | 32.12 | 28.47 | 63.89 | 76.34 | 24.63 | 33.27 |

5 rows × 505 columns

Figure 1: Price Data

### 3.2.2 Fix Null Data

Second, check the null data, drop the stocks that have too much missing data and fill out the rest null data.

Listing 2: Fill out null data

```
data.isnull().values.any()
print('Data-Shape-before-cleaning-=', data.shape)
missing_percentage = data.isnull().mean().sort_values(ascending=False)
missing_percentage.head(10)
# drop the stock which has too much missing data!
dropped_list = sorted(list(missing_percentage[missing_percentage > 0.03].index))
data.drop(labels=dropped_list, axis=1, inplace=True)
print('Data-Shape-after-cleaning-=', data.shape)
data = data.fillna(method='ffill')
```

### 3.2.3 Calculate Return and Volatility and Standard Scaling

Since we need to conduct clustering, we need to do standard scaling to our dataset.

Listing 3: Calculate Volatility and Return

```
#Calculate returns and create a data frame
average_return = data.pct_change().mean()*266
average_return = pd.DataFrame(average_return)
average_return.columns = ['returns']
#Calculate the volatility
average_return['volatility'] = data.pct_change().std()*np.sqrt(266)

from sklearn.preprocessing import StandardScaler
#Prepare the scaler
scale = StandardScaler().fit(average_return)
#Fit the scaler
scaled_data = pd.DataFrame(scale.fit_transform(average_return),\\
columns = average_return.columns, index = average_return.index)
X = scaled_data
X.head()
```

| Name | returns | volatility |
|---|---|---|
| A | -0.219806 | 0.033915 |
| AAL | 1.632907 | 1.548930 |
| AAP | -0.242862 | 0.789043 |
| AAPL | 0.558871 | -0.159148 |
| ABBV | 1.176459 | 0.332484 |

Figure 2: Return and Volatility

### 3.2.4 Splitting Training data and Backtest Data

To create and test our trading strategy, we need to divide the data into two groups. One group will be used for clustering to identify trading pairs, while the other group will be set aside for testing our strategy.

We will pick data from the initial four years for our training set, and allocate data from the last two years for our backtesting set.

## 3.3 Clustering

### 3.3.1 K-Means Clustering

To conduct K-means, we first start with the elbow method to find the best cluster number (k) by calculating distortion (average squared distance from cluster centers) and inertia (sum of squared distances from features to the nearest cluster center).

Listing 4: Elbow method for K-means

```
# try different k
inertia_values = []
for k in range(1, 20):
    kmeans = KMeans(n_clusters=k, random_state=100)
```

```
    kmeans.fit(scaled_data)
    inertia_values.append(kmeans.inertia_)
# elbow plot
plt.plot(range(1, 20), inertia_values, marker='o')
plt.title('Elbow Method for Optimal Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.show()
```

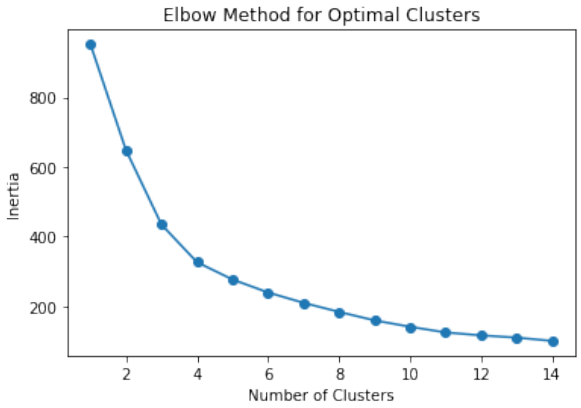


Figure 3: Kmeans Elbow Method

By observing the chart we can conclude that the optimal number of clusters would be somewhere between 5 and 6. At the iterations after 6, the model start obtaining less informative clusters.

Listing 5: K-means

```
# try different k
inertia_values = []
for k in range(1, 20):
    kmeans = KMeans(n_clusters=k, random_state=100)
    kmeans.fit(scaled_data)
    inertia_values.append(kmeans.inertia_)
# elbow plot
plt.plot(range(1, 20), inertia_values, marker='o')
plt.title('Elbow Method for Optimal Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.show()
```

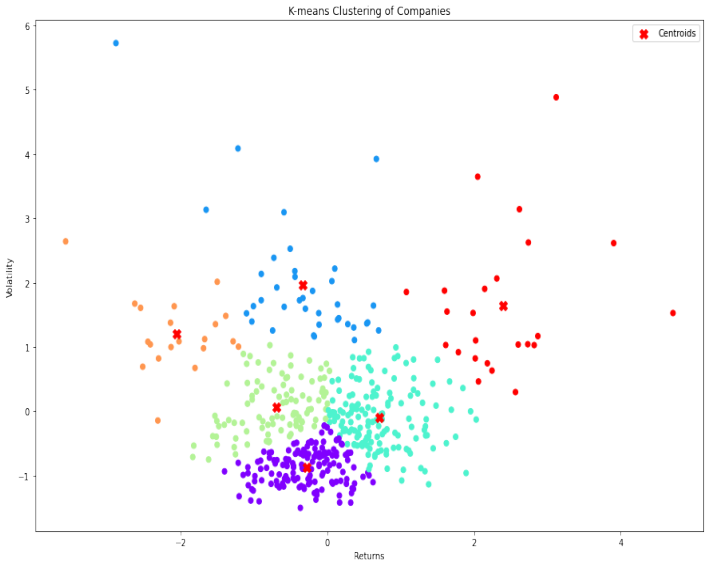


Figure 4: Kmeans Clustering

Therefore, all stocks are clustered into 6 groups and we can see the orange group and the red group consist of mainly outliers.

### 3.3.2 Hierarchical Clustering

To minimize variance distance between clusters, we'll use Ward's linkage. We can proceed to code the linkage calculation and dendrogram plotting.

Listing 6: Hierarchical Clustering Dendrograms

```
from sklearn.cluster import AgglomerativeClustering
import scipy.cluster.hierarchy as shc
# Assuming X is your data
plt.title("Hierarchical Clustering Dendrograms")
dend = shc.dendrogram(shc.linkage(X, method='ward'))
plt.xticks(rotation=90)
plt.show()
```

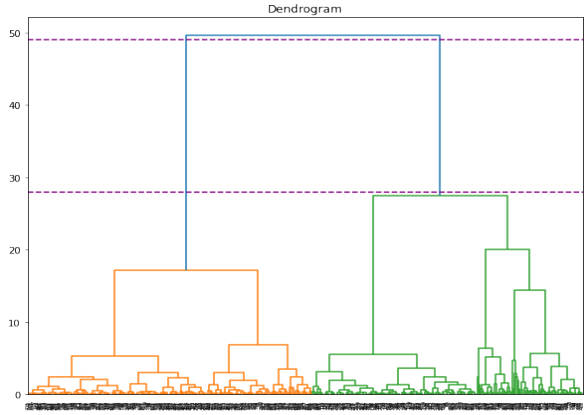


Figure 5: Hierarchical Clustering Dendrograms

The guiding principle is to select the number of clusters based on the Dendrogram's largest vertical jump, indicating the most significant distance change; thus, we opt for 2 clusters.

Listing 7: Hierarchical Clustering

```
#Fit the model
optimal_clusters = 2
agg_clustering = AgglomerativeClustering(n_clusters=optimal_clusters)
hier = agg_clustering.fit_predict(scaled_data)
#Plot the results
fig = plt.figure(figsize=(15,10))
ax = fig.add_subplot(111)
scatter = ax.scatter(X.iloc[:,0], X.iloc[:,1], c=hier, cmap='rainbow')
ax.set_title('Hierarchical Clustering of Companies')
ax.set_xlabel('Mean Return')
ax.set_ylabel('Volatility')
plt.colorbar(scatter)
plt.show()
```

And we get our Hierarchical Clustering result:

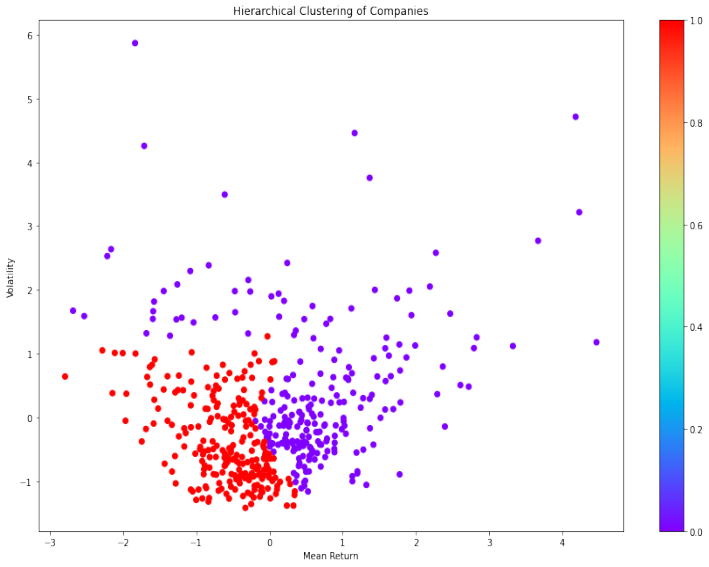


Figure 6: Hierarchical Clustering

### 3.3.3 DBSCAN

DBSCAN does not require one to specify the number of clusters in the data a prior, and we need to select the best pair of parameters by comparing the Silhouette Score.

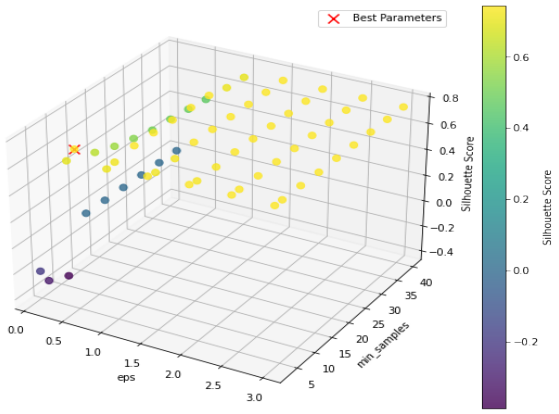


Figure 7: DBSCAN Parameter Selection

From the 3D plot above, we can select the best parameter for the DBSCAN model.

Listing 8: Hierarchical Clustering

```
# Using DBSCAN for clustering
dbscan = DBSCAN(eps=0.5, min_samples=5)
# the best paras chosen
df['cluster'] = dbscan.fit_predict(scaled_data)
dbscan_labels = dbscan.fit_predict(scaled_data)
#Plot the results
fig = plt.figure(figsize=(15,10))
ax = fig.add_subplot(111)
scatter = ax.scatter(X.iloc[:,0], X.iloc[:,1], c=df['cluster'], cmap='rainbow')
ax.set_title('DBSCAN Clustering of Companies')
ax.set_xlabel('Mean Return')
ax.set_ylabel('Volatility')
plt.colorbar(scatter)
plt.show()
```

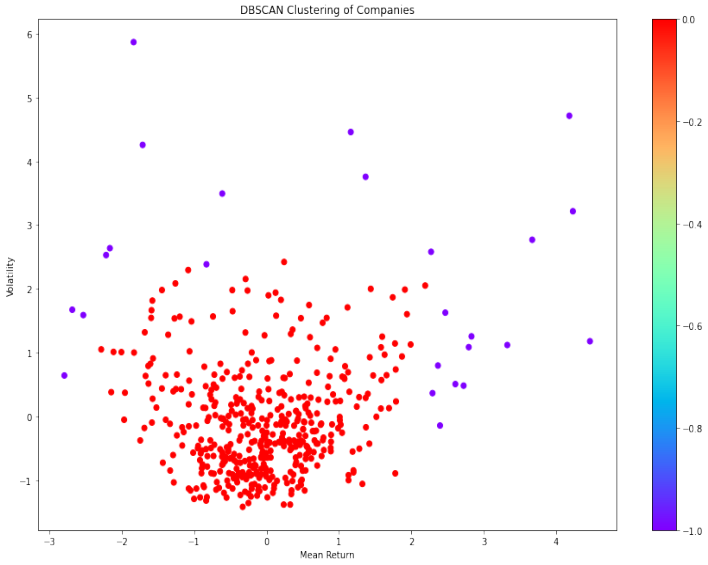


Figure 8: DBSCAN Clustering

From the above result wen can see that DBSCAN is well-suited for dealing with outliers and noisy datasets.

### 3.3.4 Affinity Propagation Clustering

First, we employ grid search to fine-tune the two primary parameters (Damping Factor,Preference) of the Affinity Propagation Clustering model, followed by training the model.

Listing 9: Grid Search

```
# grid search for best paras
for preference in preference_values:
    for damping in damping_values:
        ap = AffinityPropagation(preference=preference, damping=damping)
        labels = ap.fit_predict(X)
        silhouette_avg = silhouette_score(X, labels)

        if silhouette_avg > best_silhouette_score:
            best_silhouette_score = silhouette_avg
            best_preference = preference
            best_damping = damping
```

The best parameters after grid search:
Damping Factor = 0.8, Preference = -20

Listing 10: Affinity Propagation Clustering

```
# use the best paras
best_ap = AffinityPropagation(preference=best_preference, damping=best_damping)
best_labels = best_ap.fit_predict(X)
```

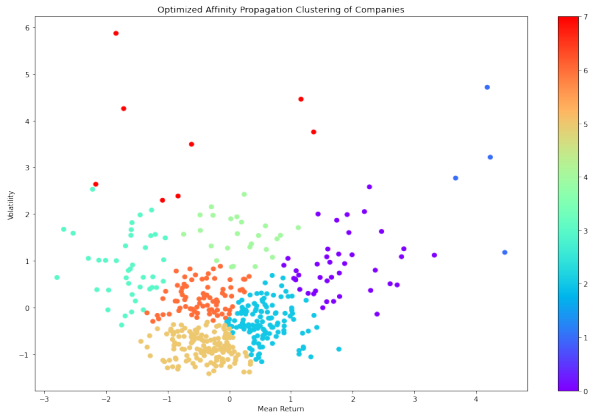


Figure 9: Affinity Propagation Clustering

Affinity Propagation Clustering has resulted in a large amount of clusters. We can identify and organize them systematically for clarity by extracting center indices and labels and plotting.

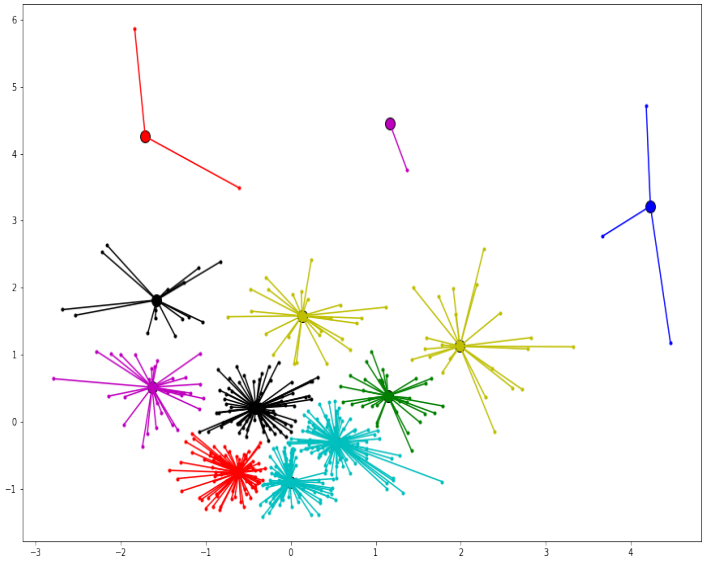

Figure 10: Affinity Propagation Clustering with Center Indices and Labels

### 3.3.5 Evaluate and Compare Clustering Models

As the clustering models are unsupervised, meaning that we don't have the labels, therefore, we can choose to compare the models by their silhouette score.

Listing 11: Evaluate Clustering Model

```
from sklearn import metrics
from sklearn.metrics import silhouette_score
print("k-Means Clustering", metrics.silhouette_score(X, \\
kmeans.fit_predict(X), metric='euclidean'))
print("Hierarchical Clustering", metrics.silhouette_score(X, \\
hc.fit_predict(X), metric='euclidean'))
print("DBSCAN Clustering", metrics.silhouette_score(X, \\
dbscan.fit_predict(X), metric='euclidean'))
print("Affinity Propagation Clustering", metrics.silhouette_score(X, \\
ap.labels_, metric='euclidean'))
```

And we get the result:

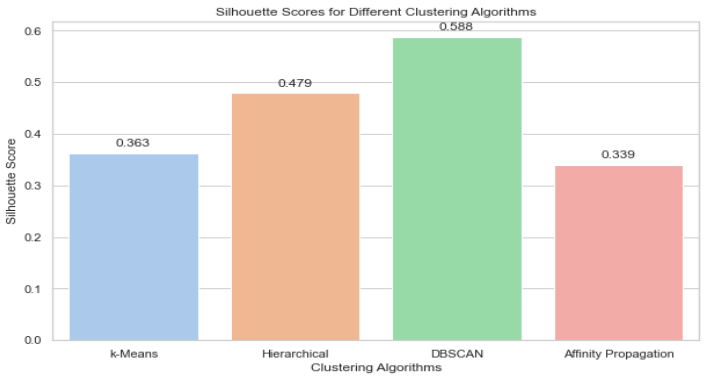


Figure 11: Compare Clustering Models

DBSCAN Clustering algorithm performed the best, and we go with it.

### 3.4 Identifying the Trading Pairs

In order to extract the trading pairs, we need to check how many trading pairs are there to be evaluated. The evaluation will perform a statistical analysis to find pairs that are cointegrated.

Pairs are deemed as cointegrated when they aren't stationary and tend to move together. Therefore, we set up a function that finds the cointegrated pairs within a cluster.

Listing 12: Detecting Trading Pairs

```
def find_coint_pairs(data, significance=0.05):
    n = data.shape[1]
    score_matrix = np.zeros((n, n))
    pvalue_matrix = np.ones((n, n))
    keys = data.keys()
    pairs = []
    for i in range(1):
        for j in range(i+1, n):
            S1 = data[keys[i]]
            S2 = data[keys[j]]
            result = coint(S1, S2)
            score = result[0]
            pvalue = result[1]
            score_matrix[i, j] = score
            pvalue_matrix[i, j] = pvalue
            if pvalue < significance:
                pairs.append((keys[i], keys[j]))
    return score_matrix, pvalue_matrix, pairs
```

And here is several trading pairs we extracted:

| **stock1_1** | **stock1_2** |
|---|---|
| A | DOV |
| A | EMN |
| A | ETN |
| A | SNI |

Now, we can proceed to backtesting these pairs to evaluate their performance by proposing pair trading strategies.

## 4 Proposing a Trading Strategy

Pair trading involves exploiting the spread between two correlated assets by establishing thresholds, developing entry and exit strategies.

As the pairs we selected exhibit a historical correlation, we can set thresholds for when the spread deviates significantly from its historical mean. These thresholds act as signals for potential entry or exit points for trades, and hopefully result in better return.

Trading signals can be built based on the identified thresholds such as historical volatility or just a simple threshold we conclude from the past.

Here, we propose 2 pair trading strategies:

### 4.1 Simple Threshold Strategy

The first one sets simple threshold, the backtesting code can be found in reference as it is too long. Taking A and DOV as an example:

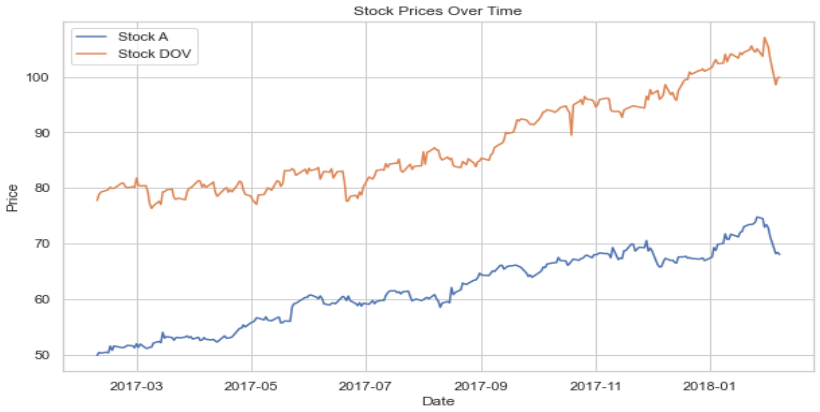


Figure 12: Stock Price

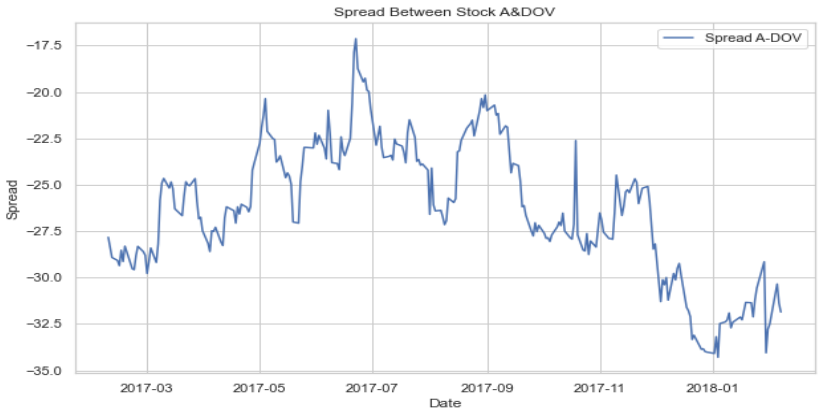


Figure 13: Spread

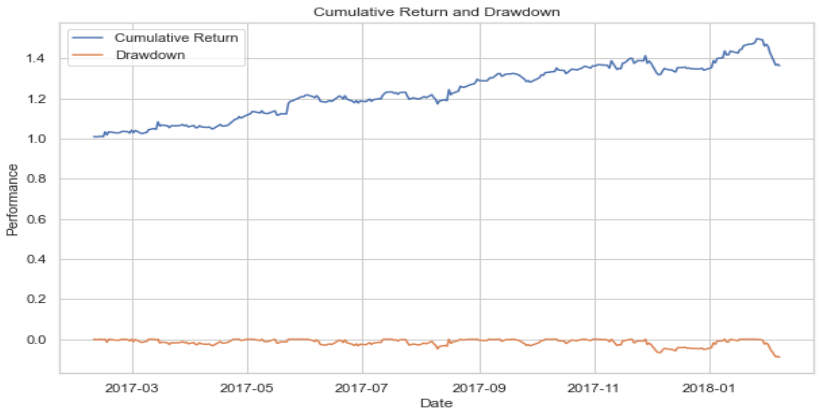


Figure 14: Cumulative Return

This straightforward strategy is effective as it generates consistent returns while effectively managing the risk with well-controlled maximum drawdown.

## 4.2 Bollinger Bands-based Strategy

The second one is Bollinger Bands-based strategy, Bollinger Bands extract signals from variance by dynamically adjusting to the volatility of the spread we are trading on. Taking ETN and A as an

| Strategy 1: Performance Metric | Value |
|---|---|
| Maximum Drawdown | -0.0904 |
| Annualized Return | 0.3631 |
| Annualized Volatility | 0.1734 |
| Sharpe Ratio | 1.9787 |

Table 2: Performance Metrics

example:

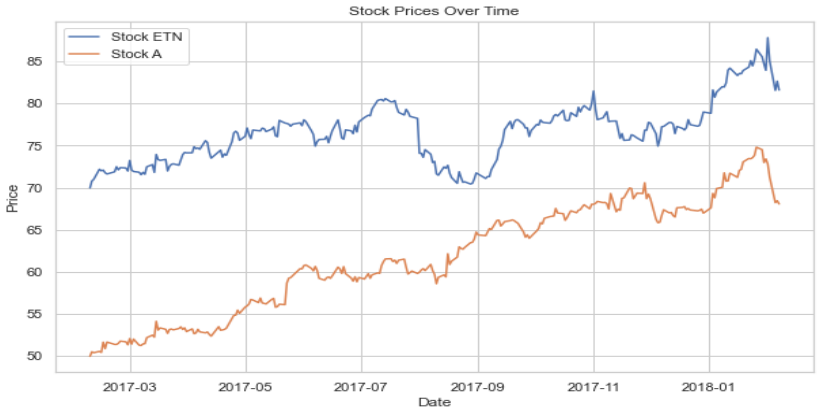


Figure 15: Stock Price

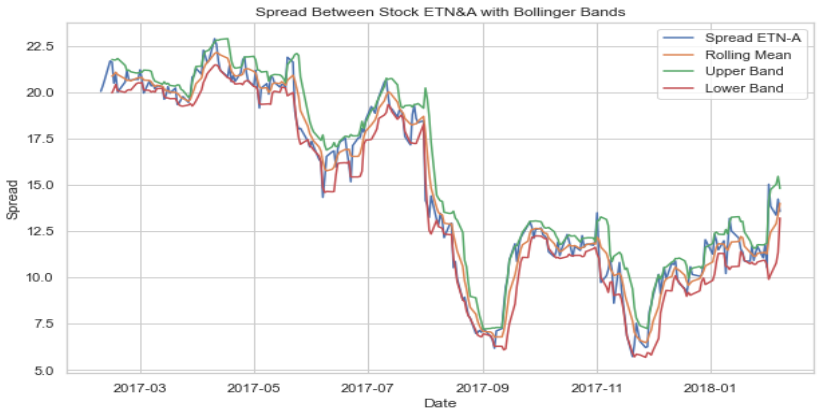


Figure 16: Spread

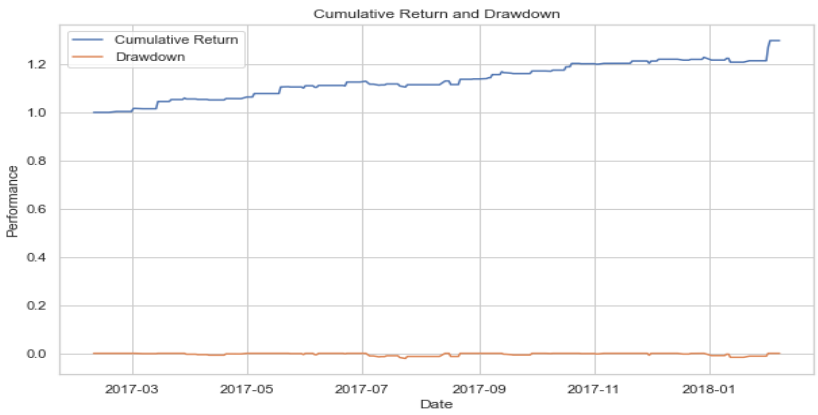


Figure 17: Cumulative Return

| **Strategy 2: Performance Metric** | **Value** |
|---|---|
| Maximum Drawdown | -0.0214 |
| Annualized Return | 0.2981 |
| Annualized Volatility | 0.0849 |
| Sharpe Ratio | 3.3767 |

Table 3: Performance Metrics

The Bollinger Bands-based strategy excels in generating continuous returns and exhibits significant improvement in minimizing the maximum drawdown. This success can be attributed to its ability to adapt to continuously changing volatility, allowing for more responsive and refined trading decisions.

# 5 Conclusion and Further Ideas

Our clustering methods work well with the pair trading strategy, helping us efficiently identify suitable pairs for trading. Using unsupervised learning to find trading pairs not only speeds up the identification process but also provides additional evidence to support our decisions. This method not only makes it easier to find appropriate pairs but also strengthens our ability to detect them.

To dive deeper into this topic, we can work on these aspects:

1. Improve the trading signals by introducing more information, for example: fundamental data.
2. Combine clustering with monte carlo method to gather more historical data and reduce the variance in our detection.
3. Implement rolling clustering and continuously adjust the strategy.

# References

@onlinealgotrading101, title = Cluster Analysis Guide, author = AlgoTrading101, year = Year, url = https://algotrading101.com/learn/cluster-analysis-guide/,

@onlinewangy8989, title = Pairs Trading with Machine Learning, author = Wangy8989, year = Year, url = https://github.com/wangy8989/Pairs-Trading-with-Machine-Learning,

@onlineigorwounds, title = Cluster Analysis Machine Learning for Pairs Trading, author = Igor-Wounds, year = Year, url = https://github.com/IgorWounds/Cluster-Analysis-Machine-Learning-for-Pairs-Trading/tree/main,

@onlinehudsonthames, title = Employing Machine Learning for Trading Pairs Selection, author = HudsonThames, year = Year, url = https://hudsonthames.org/employing-machine-learning-for-trading-pairs-selection/,

# A Pair Trading Strategy: Simple Threshold

Listing 13: Pair Trading Strategy: Simple Threshold

```
def backtestor(df,i,j):
    # Calculate the spread between the two stocks
    first = i
    second = j
    df['Spread'] = df[first] - df[second]

    # Define entry and exit points based on the spread
    entry_threshold = 1.0  # Adjust as needed
    exit_threshold = 0.5   # Adjust as needed

    df['Long_Entry'] = np.where(df['Spread'] < -entry_threshold, 1, 0)
    df['Short_Entry'] = np.where(df['Spread'] > entry_threshold, 1, 0)
    df['Exit'] = np.where(np.abs(df['Spread']) < exit_threshold, 1, 0)

    # Calculate daily returns
    df['Return_'+first] = df[first].pct_change()
    df['Return_'+second] = df[second].pct_change()

    # Calculate strategy returns
    df['Strategy_Return'] = df['Long_Entry'].shift(1) * \\
    df['Return_'+first] - df['Short_Entry'].shift(1) * df['Return_'+second]

    # Calculate cumulative returns
    df['Cumulative_Return'] = (1 + df['Strategy_Return']).cumprod()

    # Calculate drawdown
    df['Previous_Peak'] = df['Cumulative_Return'].cummax()
    df['Drawdown'] = (df['Cumulative_Return'] / df['Previous_Peak']) - 1

    # Calculate Sharpe ratio (assuming risk-free rate is 0 for simplicity)
    sharpe_ratio = df['Strategy_Return'].mean() / df['Strategy_Return'].std()

    # Plotting
    plt.figure(figsize=(10, 6))
    plt.plot(df.index, df[first], label='Stock '+first)
    plt.plot(df.index, df[second], label='Stock '+second)
    plt.title('Stock Prices Over Time')
    plt.xlabel('Date')
    plt.ylabel('Price')
    plt.legend()
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(df.index, df['Spread'], label='Spread '+first+'-'+second)
    '''
    plt.axhline(y=entry_threshold, color='r', linestyle='--', label='Entry Threshold')
    plt.axhline(y=-entry_threshold, color='r', linestyle='--')
    plt.axhline(y=exit_threshold, color='g', linestyle='--', label='Exit Threshold')
    plt.axhline(y=-exit_threshold, color='g', linestyle='--')
    '''
    plt.title('Spread Between Stock '+first+'&'+second)
    plt.xlabel('Date')
    plt.ylabel('Spread')
```

```
    plt.legend()
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(df.index, df['Cumulative_Return'], label='Cumulative Return')
    plt.plot(df.index, df['Drawdown'], label='Drawdown')
    plt.title('Cumulative Return and Drawdown')
    plt.xlabel('Date')
    plt.ylabel('Performance')
    plt.legend()
    plt.show()

    # Calculate maximum drawdown
    max_drawdown = df['Drawdown'].min()
    # Calculate annualized return and annualized volatility
    (252 trading days in a year for simplicity)
    trading_days_per_year = 252
    annualized_return = (df['Cumulative_Return'].iloc[-1] **
    (trading_days_per_year / len(df))) - 1
    annualized_volatility = df['Strategy_Return'].std() * np.sqrt(trading_days_per_year)
    # Print additional performance metrics
    print(f'Maximum Drawdown: {max_drawdown:.4f}')
    print(f'Annualized Return: {annualized_return:.4f}')
    print(f'Annualized Volatility: {annualized_volatility:.4f}')
    print(f'Sharpe Ratio: {sharpe_ratio:.10f}')
```

# B Pair Trading Strategy: Bollinger Bands-based strategy

Listing 14: Bollinger Bands-based strategy

```
def backtestor_Bollinger(df, i, j, window=5, num_std_dev=1):
    # Calculate the spread between the two stocks
    first = i
    second = j
    df['Spread'] = df[first] - df[second]

    # Calculate rolling mean and standard deviation of the spread
    df['Rolling_Mean'] = df['Spread'].rolling(window=window).mean()
    df['Upper_Band'] = df['Rolling_Mean'] + num_std_dev
    * df['Spread'].rolling(window=window).std()
    df['Lower_Band'] = df['Rolling_Mean'] - num_std_dev
    * df['Spread'].rolling(window=window).std()

    # Define entry and exit points based on Bollinger Bands
    df['Long_Entry'] = np.where(df['Spread'] < df['Lower_Band'], 1, 0)
    df['Short_Entry'] = np.where(df['Spread'] > df['Upper_Band'], 1, 0)
    df['Exit'] = np.where((df['Spread'] > df['Lower_Band'])
    & (df['Spread'] < df['Upper_Band']), 1, 0)

    # Calculate daily returns
    df['Return_'+first] = df[first].pct_change()
    df['Return_'+second] = df[second].pct_change()

    # Calculate strategy returns
    df['Strategy_Return'] = df['Long_Entry'].shift(1)
    * df['Return_'+first] - df['Short_Entry'].shift(1) * df['Return_'+second]
```

```
# Calculate cumulative returns
df['Cumulative_Return'] = (1 + df['Strategy_Return']).cumprod()

# Calculate drawdown
df['Previous_Peak'] = df['Cumulative_Return'].cummax()
df['Drawdown'] = (df['Cumulative_Return'] / df['Previous_Peak']) - 1

# Calculate Sharpe ratio (assuming risk-free rate is 0 for simplicity)
sharpe_ratio = df['Strategy_Return'].mean() / df['Strategy_Return'].std()

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(df.index, df[first], label='Stock-'+first)
plt.plot(df.index, df[second], label='Stock-'+second)
plt.title('Stock-Prices-Over-Time')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(df.index, df['Spread'], label='Spread-'+first+'-'+second)
plt.plot(df.index, df['Rolling_Mean'], label='Rolling-Mean')
plt.plot(df.index, df['Upper_Band'], label='Upper-Band')
plt.plot(df.index, df['Lower_Band'], label='Lower-Band')
plt.title('Spread-Between-Stock-'+first+'&'+second + '-with-Bollinger-Bands')
plt.xlabel('Date')
plt.ylabel('Spread')
plt.legend()
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(df.index, df['Cumulative_Return'], label='Cumulative-Return')
plt.plot(df.index, df['Drawdown'], label='Drawdown')
plt.title('Cumulative-Return-and-Drawdown')
plt.xlabel('Date')
plt.ylabel('Performance')
plt.legend()
plt.show()

# Calculate maximum drawdown
max_drawdown = df['Drawdown'].min()

# Calculate annualized return and annualized volatility (252 trading days in a year
trading_days_per_year = 252
annualized_return = (df['Cumulative_Return'].iloc[-1]
** (trading_days_per_year / len(df))) - 1
annualized_volatility = df['Strategy_Return'].std()
* np.sqrt(trading_days_per_year)

# Print additional performance metrics
print(f'Maximum-Drawdown:-{max_drawdown:.4f}')
print(f'Annualized-Return:-{annualized_return:.4f}')
print(f'Annualized-Volatility:-{annualized_volatility:.4f}')
print(f'Sharpe-Ratio:-{sharpe_ratio:.10f}')
```