Final Report: Enhanced Pairs Trading Strategy with Risk Management and Kalman Filtering

Overview

This project aims to develop a robust pairs trading strategy by integrating risk management techniques and Kalman filtering, building upon the work done in the previous two weeks. Pairs trading involves identifying two assets with a historical correlation and exploiting temporary divergences in their price relationship. The core idea is to simultaneously take a long position in the undervalued asset and a short position in the overvalued asset, betting on the convergence of their prices.

In the first week, the basic pairs trading algorithm was implemented, replacing the traditional Z-score with Bollinger Bands to generate buy and sell signals. The second week focused on incorporating a Kalman filter to dynamically adjust the hedge ratio and optimize the trading strategy's parameters. This final stage integrates risk management techniques, particularly position sizing, to further enhance the strategy's robustness and performance. The goal is to control risk while maintaining profitability by applying stop-loss orders, dynamic position sizing based on volatility, and diversifying across multiple pairs of stocks.

Week 1: Pairs Trading with Bollinger Bands

In the initial implementation of the pairs trading strategy, the Z-score method for identifying trading signals was replaced with Bollinger Bands. Bollinger Bands consist of a middle band representing a moving average of the price and two outer bands that are a certain number of standard deviations away from the moving average.

The primary reasons for choosing Bollinger Bands over the Z-score are:

- **Dynamic Volatility Assessment:** Bollinger Bands adjust dynamically to changes in market volatility. The width of the bands expands during periods of high volatility and contracts during periods of low volatility. This adaptive nature allows for more responsive and accurate trading signals compared to the static Z-score.
- **Clearer Overbought/Oversold Levels:** Bollinger Bands provide clear levels for identifying overbought and oversold conditions. When the price touches or exceeds the upper band, it signals an overbought condition and a potential short opportunity. Conversely, when the price touches or falls below the lower band, it signals an oversold condition and a potential long opportunity.
- **Reduced Sensitivity to Outliers:** The Z-score can be sensitive to outliers, leading to false signals. Bollinger Bands, by considering the moving average and standard

deviation, offer a smoother representation of price movements, reducing the impact of outliers.

Week 2: Kalman Filtering for Dynamic Hedge Ratio Adjustment

In the second week, a Kalman filter was introduced to dynamically adjust the hedge ratio between the two stocks in a pair.

What is a Kalman Filter?
The Kalman filter is a recursive algorithm used to estimate the state of a dynamic system from a series of incomplete and noisy measurements. It is particularly useful in financial time series analysis because it can adapt to changing market conditions and provide optimal estimates of unobserved variables, such as the hedge ratio in pairs trading.

Why Kalman Filter over Rolling Averages?
The Kalman filter offers several advantages over simple rolling averages:

- **Adaptability:** Unlike rolling averages, which treat all data points within the window equally, the Kalman filter assigns different weights to measurements based on their uncertainty. This allows the filter to quickly adapt to sudden changes in the relationship between the two stocks.
- **Optimal Estimation:** The Kalman filter provides the best linear unbiased estimate of the system's state by minimizing the mean square error. This results in a more accurate and responsive hedge ratio compared to rolling averages.
- **Noise Reduction:** The Kalman filter effectively filters out noise from the measurements, providing a smoother and more reliable estimate of the hedge ratio. This is particularly important in volatile markets where noise can lead to false trading signals.

The Kalman filter estimates the spread in real-time and adjusts the hedge ratio accordingly, minimizing risk exposure while maximizing returns. The trade function was then backtested with optimized parameters for the Bollinger Bands, such as window size and the number of standard deviations (k).

Final Project: Integrating Risk Management via Position Sizing

The final project integrates risk management techniques, focusing primarily on position sizing, into the existing pairs trading model.

Why Position Sizing?
Position sizing is a critical aspect of risk management in trading. It involves determining the

appropriate amount of capital to allocate to each trade based on the trader's risk tolerance, account size, and the characteristics of the trading strategy. Proper position sizing helps to:

- **Control Risk:** By limiting the amount of capital at risk on any single trade, position sizing prevents large losses from eroding the trading account.
- **Optimize Returns:** Position sizing allows traders to take advantage of high-probability trading opportunities while minimizing risk exposure.
- **Ensure Consistency:** Consistent position sizing helps to maintain a stable risk profile and avoid impulsive or emotional trading decisions.

Implementation
The implementation of risk management in this project involves the following steps:

1. **Stop-Loss Orders:** A threshold for the spread between two stocks is defined. If the spread exceeds this threshold, the position is exited to limit losses.
2. **Position Sizing:** The position size is calculated based on the volatility of the spread. A larger position can be taken when the spread is stable (low volatility), and a smaller position is taken when volatility is high.
3. **Portfolio Diversification:** Rather than trading a single pair of stocks, multiple pairs are traded to diversify the portfolio, reducing the overall risk exposure.
4. **Kalman Filtering:** The Kalman filter is used to dynamically adjust the hedge ratio between the two stocks in a pair. The Kalman filter allows real-time estimation of the spread and adjusts the hedge ratio accordingly, minimizing risk exposure while maximizing returns.

```python
import yfinance as yf
import pandas as pd
import numpy as np
import statsmodels.api as sm
from pykalman import KalmanFilter
import matplotlib.pyplot as plt
from datetime import datetime

start = datetime(2010, 1, 1)
end = datetime(2020, 1, 1)

tickers = ['ADBE','MSFT']

df = yf.download(tickers, start, end)['Close']
df = df.dropna()
```

```
ratio=df['ADBE']/df['MSFT']
train_test_ratio= round(len(ratio)*0.7)
print(train_test_ratio)

test_start= round(len(ratio)*0.7)
val_start= round(len(ratio)*0.9)
print(test_start)
print(val_start)
print(len(ratio))

train= ratio[:train_test_ratio]
test= ratio[train_test_ratio:val_start]
OS= ratio[val_start:]

x = df['ADBE']
y = df['MSFT']

train_x = x[:test_start]
train_y = y[:test_start]

test_x = x[test_start:val_start]
test_y = y[test_start:val_start]

val_x = x[val_start:]
val_y = y[val_start:]


plt.plot(train)
plt.show()
```

Explanation:

1. **Import Libraries:** This section imports the necessary Python libraries:
   - `yfinance`: To download financial data from Yahoo Finance.
   - `pandas`: For data manipulation and analysis using DataFrames.
   - `numpy`: For numerical computations.
   - `statsmodels.api`: For statistical modeling, particularly for linear regression.
   - `pykalman`: For implementing the Kalman filter.

    - `matplotlib.pyplot`: For creating visualizations.
    - `datetime`: For working with dates.
2. **Define Time Period and Tickers:**
    - `start` and `end` variables define the start and end dates for the data retrieval (January 1, 2010, to January 1, 2020).
    - `tickers` list contains the stock symbols for the pairs trading strategy (Adobe - ADBE and Microsoft - MSFT).
3. **Download Data:**
    - `yf.download(tickers, start, end)['Close']` downloads the historical closing prices for the specified tickers within the defined time period from Yahoo Finance. The `['Close']` part selects only the closing prices.
    - `df = df.dropna()` removes any rows with missing data (NaN values) to ensure data quality.
4. **Calculate Ratio and Split Data:**
    - `ratio=df['ADBE']/df['MSFT']` calculates the ratio of the closing prices of Adobe to Microsoft. This ratio is the basis for the pairs trading strategy. The assumption is that this ratio tends to revert to its mean.
    - The code then splits the data into training, testing, and out-of-sample (OS) sets. This is crucial for developing and evaluating the trading strategy.
    - `train_test_ratio = round(len(ratio) * 0.7)` calculates the index to split the data into 70% training and 30% combined testing and out-of-sample.
    - `test_start = round(len(ratio) * 0.7)` and `val_start = round(len(ratio) * 0.9)` calculate the indices for splitting the remaining 30% into testing (20%) and out-of-sample (10%) sets.
    - The `train`, `test`, and `OS` variables store the corresponding ratio data for each set.
    - The code also creates separate `x` and `y` variables for the prices of ADBE and MSFT, respectively, and splits them into training, testing, and validation sets similar to the ratio.
5. **Plot Training Ratio:**
    - `plt.plot(train)` plots the ratio of the stock prices in the training set.
    - `plt.show()` displays the plot. This allows for visual inspection of the ratio's behavior during the training period.

Why It Works:

- **Data Acquisition:** `yfinance` provides a convenient way to access historical stock data, which is essential for backtesting and developing trading strategies.
- **Ratio Calculation:** The ratio of two correlated stocks is the foundation of pairs trading. The strategy bets on the mean reversion of this ratio.
- **Data Splitting:** Splitting the data into training, testing, and out-of-sample sets is crucial for avoiding overfitting and evaluating the strategy's performance on unseen data.

- **Visualization:** Plotting the ratio helps to understand its patterns and volatility, which are important for parameter tuning and risk management.

Relevance to Project:

This initial block sets the stage for the entire project by:

- Acquiring the necessary data.
- Calculating the key ratio for pairs trading.
- Splitting the data to allow for proper training, testing, and validation of the strategy.
- Providing a visual overview of the ratio's behavior.

```
def KalmanFilterAverage(x):
    # Construct a Kalman filter
    kf = KalmanFilter(transition_matrices = [1],
                      observation_matrices = [1],
                      initial_state_mean = 0,
                      initial_state_covariance = 1,
                      observation_covariance=1,
                      transition_covariance=.01)

    # Use the observed values of the price to get a rolling mean
    state_means, _ = kf.filter(x.values)
    state_means = pd.Series(state_means.flatten(), index=x.index)
    return state_means
```

Explanation:

1. `KalmanFilterAverage(x)` **Function:** This function takes a time series `x` (e.g., stock prices) as input and applies a Kalman filter to smooth the data. In this case, the Kalman filter is configured to estimate a *local level* or *time-varying mean* of the series.
2. **Kalman Filter Initialization:**
   - `kf = KalmanFilter(...)`: This line initializes the Kalman filter object. Let's break down the parameters:
     - `transition_matrices=`: This defines the state transition matrix. Here, it's a simple ``, meaning the filter assumes the underlying state (the mean) is constant over time but can drift randomly. The equation is: `state(t+1) = 1 * state(t) + noise`.
     - `observation_matrices=`: This defines the observation matrix. It's also ``, meaning the observed value (the stock price) is directly related to the

underlying state (the mean). The equation is: `observation(t) = 1 * state(t) + noise`.

        - `initial_state_mean=0`: The filter's initial guess for the state (the mean) is 0. This will quickly be updated as it sees data.
        - `initial_state_covariance=1`: The filter's initial uncertainty about the state is 1. A higher value means the filter will trust the initial observations more.
        - `observation_covariance=1`: This defines the noise in the *observation*. It represents how much noise there is in the stock price itself. A higher value means the filter trusts the observations less.
        - `transition_covariance=.01`: This defines the noise in the *state transition*. It represents how much the underlying mean is allowed to change at each step. A smaller value means the filter assumes the mean is more stable.

3. **Filtering:**
    - `state_means, _ = kf.filter(x.values)`: This is the core of the Kalman filter. It applies the filter to the input data `x.values`. The `filter` method returns the estimated state means (the smoothed values) and the state covariances (uncertainty about the state). We only care about the means here, so the covariances are discarded using `_`.
4. **Output:**
    - `state_means = pd.Series(state_means.flatten(), index=x.index)`: The `state_means` are converted back into a Pandas Series, using the original index of the input data `x`. This makes it easier to work with the smoothed data.
    - `return state_means`: The function returns the smoothed time series.

Why It Works:

The Kalman filter works by recursively updating its estimate of the state (the mean) based on new observations. It balances two sources of information:

- **The Process Model (transition matrices and transition covariance):** This describes how the state is expected to evolve over time. In this case, it assumes the mean is relatively constant.
- **The Measurement Model (observation matrices and observation covariance):** This describes how the observations (stock prices) are related to the state. It also accounts for noise in the observations.

The filter combines these two sources of information to produce an optimal estimate of the state.

Relevance to Project:

This function is used to smooth the stock price data before it's fed into the Kalman filter regression. Smoothing helps to reduce noise and improve the accuracy of the hedge ratio estimation. This leads to more stable and reliable trading signals.

```
#  Kalman filter regression
def KalmanFilterRegression(x,y):

    delta = 1e-3
    trans_cov = delta / (1 - delta) * np.eye(2) # How much random walk
wiggles
    obs_mat = np.expand_dims(np.vstack([[x], [np.ones(len(x))]]).T,
axis=1)

    kf = KalmanFilter(n_dim_obs=1, n_dim_state=2, # y is 1-dimensional,
(alpha, beta) is 2-dimensional
                      initial_state_mean=[0,0],
                      initial_state_covariance=np.ones((2, 2)),
                      transition_matrices=np.eye(2),
                      observation_matrices=obs_mat,
                      observation_covariance=2,
                      transition_covariance=trans_cov)

    # Use the observations y to get running estimates and errors for the
state parameters
    state_means, state_covs = kf.filter(y.values)
    return state_means
```

Explanation:

1. **`KalmanFilterRegression(x, y)` Function:** This function implements Kalman filter-based linear regression. It estimates the relationship between two time series, `x` (independent variable) and `y` (dependent variable), where the coefficients of the regression are allowed to vary over time. This is more sophisticated than a standard linear regression because it assumes the relationship between the assets can change dynamically.
2. **Defining Parameters:**

    - `delta = 1e-3`: This parameter controls the stability of the regression coefficients. A smaller `delta` means the coefficients are assumed to be more stable over time.
    - `trans_cov = delta / (1 - delta) * np.eye(2)`: This calculates the transition covariance matrix, which defines how much the regression coefficients (alpha and beta) are allowed to change at each time step. The `np.eye(2)` creates a 2x2 identity matrix, ensuring that alpha and beta can change independently. The smaller the `delta`, the smaller the `trans_cov`, and the more stable the coefficients.
    - `obs_mat = np.expand_dims(np.vstack([[x], [np.ones(len(x))]]).T, axis=1)`: This creates the observation matrix. This matrix links the *state* (alpha and beta) to the *observation* (y). Let's break it down:
        - `np.ones(len(x))`: Creates a vector of ones with the same length as `x`. This represents the intercept term in the regression.
        - `np.vstack([[x], [np.ones(len(x))]])`: Stacks the `x` values and the vector of ones vertically, creating a 2xN matrix.
        - `.T`: Transposes the matrix, so it's now an Nx2 matrix.
        - `np.expand_dims(..., axis=1)`: Adds an extra dimension to the matrix, making it an Nx1x2 matrix. This is required by the `pykalman` library.
        - So, effectively, `obs_mat` is an Nx1x2 matrix where each row is `[[x_i, 1]]`. This allows the Kalman filter to estimate `y_i = alpha + beta * x_i` at each time step `i`.
3. **Kalman Filter Initialization:**
    - `kf = KalmanFilter(...)`: Initializes the Kalman filter for regression. Key parameters:
        - `n_dim_obs=1, n_dim_state=2`: Specifies that the observation (`y`) is 1-dimensional, and the state (alpha and beta) is 2-dimensional.
        - `initial_state_mean=`: Sets the initial guess for alpha and beta to 0.
        - `initial_state_covariance=np.ones((2, 2))`: Sets the initial uncertainty about alpha and beta to 1.
        - `transition_matrices=np.eye(2)`: Assumes that alpha and beta are constant over time (but can drift randomly).
        - `observation_matrices=obs_mat`: Uses the calculated observation matrix to link the state to the observation.
        - `observation_covariance=2`: Defines the noise in the observation (`y`). A higher value means the filter trusts the observations less.
        - `transition_covariance=trans_cov`: Uses the calculated transition covariance to define how much alpha and beta can change at each step.
4. **Filtering:**

    - `state_means, state_covs = kf.filter(y.values)`: Applies the Kalman filter to estimate the state (alpha and beta) based on the observations (`y` and `x`). The function returns the estimated state means (the smoothed alpha and beta values) and the state covariances (uncertainty about alpha and beta).
5. **Output:**
    - `return state_means`: Returns the estimated state means, which are the time-varying alpha and beta values.

Why It Works:

This function extends the basic Kalman filter to perform linear regression where the coefficients are not fixed but can evolve over time. This is crucial for pairs trading because the relationship between the two assets can change due to various market factors. By using a Kalman filter, the strategy can adapt to these changing relationships and maintain its effectiveness.

The observation matrix `obs_mat` is cleverly constructed to represent the linear regression equation within the Kalman filter framework. The transition covariance matrix allows the regression coefficients to drift randomly over time, capturing the dynamic nature of the relationship.

Relevance to Project:

This function is the heart of the dynamic hedge ratio adjustment. It provides estimates of alpha (intercept) and beta (slope) that are used to calculate the spread between the two assets. This spread is then used to generate trading signals. The dynamic adjustment of the hedge ratio is a key advantage of this strategy over simpler approaches that use fixed hedge ratios.

python
```python
def half_life(spread):
    spread_lag = spread.shift(1)
    spread_lag.iloc[0] = spread_lag.iloc[1]

    spread_ret = spread - spread_lag
    spread_ret.iloc[0] = spread_ret.iloc[1]

    spread_lag2 = sm.add_constant(spread_lag)

    model = sm.OLS(spread_ret,spread_lag2)
    res = model.fit()
    halflife = int(round(-np.log(2) / res.params[1],0))

    if halflife <= 0:
        halflife = 1
    return halflife
```

Explanation:

1. **`half_life(spread)` Function:** This function estimates the half-life of the mean reversion process in a given spread. The half-life is the number of periods it takes, on average, for the spread to revert halfway back to its mean. This is a crucial parameter for pairs trading, as it helps determine how long to hold a position.
2. **Calculate Spread Returns:**
   - `spread_lag = spread.shift(1)`: Creates a lagged version of the spread (shifted by one period).
   - `spread_lag.iloc = spread_lag.iloc`: Fills the first value of the lagged spread with the second value to avoid `NaN`.
   - `spread_ret = spread - spread_lag`: Calculates the difference between the current spread and the lagged spread. This represents the change in the spread.
   - `spread_ret.iloc = spread_ret.iloc`: Fills the first value of the spread returns with the second value to avoid `NaN`.
3. **Linear Regression:**
   - `spread_lag2 = sm.add_constant(spread_lag)`: Adds a constant term to the lagged spread. This is necessary for the linear regression.
   - `model = sm.OLS(spread_ret, spread_lag2)`: Creates an OLS (Ordinary Least Squares) regression model. The model attempts to predict the spread return based on the lagged spread and the constant term.
   - `res = model.fit()`: Fits the regression model to the data.
4. **Calculate Half-Life:**
   - `halflife = int(round(-np.log(2) / res.params, 0))`: Calculates the half-life using the coefficient of the lagged spread from the regression results. The formula `-np.log(2) / res.params` is derived from the mean-reversion process equation. `res.params` represents the coefficient on the lagged spread, which indicates the speed of mean reversion.
   - `if halflife <= 0: halflife = 1`: If the calculated half-life is zero or negative (which can happen if the coefficient is positive or very close to zero), it's set to 1 to avoid errors and ensure a reasonable value.

Why It Works:

The function is based on the following mean-reversion model:

```
spread_ret(t) = alpha + beta * spread(t-1) + error(t)
```

Where:

- `spread_ret(t)` is the change in the spread at time t.
- `spread(t-1)` is the spread at time t-1.
- `alpha` is a constant.
- `beta` is the coefficient that determines the speed of mean reversion. It's expected to be negative.
- `error(t)` is the error term.

The half-life is then derived from `beta` using the formula mentioned above.

Relevance to Project:

The half-life is a valuable parameter for setting the trading strategy's parameters. It provides an estimate of how long the spread is expected to deviate from its mean. This information can be used to set stop-loss levels, position sizes, and the holding period for trades. While not directly used in the presented code for trade execution, it's valuable for informing the choice of the `window` parameter in the `trade` function.

python
```
regression_results = KalmanFilterRegression(KalmanFilterAverage(x),
KalmanFilterAverage(y))

a_estimate = regression_results[:, 0]  # Slope (alpha)
b_estimate = regression_results[:, 1]  # Intercept (beta)

# Print the last estimates for slope and intercept
print(f"Estimated slope (a): {a_estimate[-1]:.4f}")
print(f"Estimated intercept (b): {b_estimate[-1]:.4f}")

spread = y - (a_estimate + x*b_estimate)
spread_mean =  spread.mean()
spread_std = spread.std()
```

Explanation:

1. **Calculate Regression Results:**
    - `regression_results = KalmanFilterRegression(KalmanFilterAverage(x), KalmanFilterAverage(y))`: This line calls the `KalmanFilterRegression` function, which we discussed previously. It feeds in the smoothed stock prices of

ADBE (`x`) and MSFT (`y`) using the `KalmanFilterAverage` function. The output `regression_results` contains the time-varying estimates of the regression coefficients (alpha and beta) at each time step.

2. **Extract Alpha and Beta:**
    - `a_estimate = regression_results[:, 0]`: Extracts the time-varying intercept (alpha) from the `regression_results`. Remember, the `KalmanFilterRegression` function returns a matrix where each row represents a time step and the columns represent the state variables (alpha and beta). The `[:, 0]` selects all rows and the first column (alpha).
    - `b_estimate = regression_results[:, 1]`: Extracts the time-varying slope (beta) from the `regression_results`. The `[:, 1]` selects all rows and the second column (beta).
3. **Print Estimates:**
    - `print(f"Estimated slope (a): {a_estimate[-1]:.4f}")`: Prints the last estimated value of the slope (beta), formatted to four decimal places.
    - `print(f"Estimated intercept (b): {b_estimate[-1]:.4f}")`: Prints the last estimated value of the intercept (alpha), formatted to four decimal places. These last values represent the model's best estimate of the relationship between the two stocks at the end of the training period.
4. **Calculate Spread:**
    - `spread = y - (a_estimate + x*b_estimate)`: Calculates the spread between the two stocks using the estimated alpha and beta values. This is the key to the pairs trading strategy. If you rearrange the equation, you get `y = a_estimate + b_estimate * x + spread`. The goal is to trade when the spread deviates significantly from its mean.
    - `spread_mean = spread.mean()`: Calculates the mean of the spread. This represents the "equilibrium" value of the spread.
    - `spread_std = spread.std()`: Calculates the standard deviation of the spread. This measures the volatility of the spread and is used to set trading thresholds.

Why It Works:

This code block leverages the output of the `KalmanFilterRegression` function to calculate the spread between the two stocks. The spread is calculated using the time-varying alpha and beta estimates, which allows the strategy to adapt to changing market conditions. The mean and standard deviation of the spread are then used to define trading signals based on deviations from the mean.

Relevance to Project:

This code block is crucial for generating the trading signals. The spread represents the divergence between the two stocks, and the mean and standard deviation of the spread are used to define the upper and lower bands for trading. The dynamic adjustment of alpha and beta using the Kalman filter ensures that the spread is calculated accurately, leading to more reliable trading signals. This then leads to greater profit and less loss for trades being placed.

```
k=1
movavg=train.rolling(window=20, center= False).mean()

#calculating standard deviation (volatility)
std=train.rolling(window=20, center=False).std()

upper_band= movavg + (k*std)
lower_band= movavg - (k*std)

plt.figure(figsize=(12,7))
plt.plot(ratio, label='Ratio')
buy = train.copy()
sell = train.copy()
buy[train>lower_band] = 0
sell[train<upper_band] = 0

plt.figure(figsize=(12,7))
plt.plot(train, label='ratio')

buy.plot(color='g', linestyle='None', marker='^')
sell.plot(color='r', linestyle='None', marker='^')
x1, x2, y1, y2 = plt.axis()
plt.axis((x1, x2, ratio.min(), ratio.max()))
plt.xlim('2010-01-01','2017-01-01')
plt.legend(['Train', 'Buy Signal', 'Sell Signal'])
plt.show()

plt.figure(figsize=(12,7))
S1 = df['ADBE'].iloc[:1761]
S2 = df['MSFT'].iloc[:1761]

S1[60:].plot(color='b')
S2[60:].plot(color='c')
buyR = 0*S1.copy()
sellR = 0*S1.copy()
```

```
# When you buy the ratio, you buy stock S1 and sell S2
buyR[buy!=0] = S1[buy!=0]
sellR[buy!=0] = S2[buy!=0]

# When you sell the ratio, you sell stock S1 and buy S2
buyR[sell!=0] = S2[sell!=0]
sellR[sell!=0] = S1[sell!=0]

buyR[60:].plot(color='g', linestyle='None', marker='^')
sellR[60:].plot(color='r', linestyle='None', marker='^')
x1, x2, y1, y2 = plt.axis()
plt.axis((x1, x2, min(S1.min(), S2.min()), max(S1.max(), S2.max())))
plt.ylim(25, 105)
plt.xlim('2013-03-22', '2016-07-04')

plt.legend(['ADBE', 'MSFT', 'Buy Signal', 'Sell Signal'])
plt.show()
```

Explanation:

1. **Calculate Bollinger Bands:**
   - `k = 1`: Defines the number of standard deviations for the Bollinger Bands. This determines the width of the bands.
   - `movavg = train.rolling(window=20, center=False).mean()`: Calculates the 20-day rolling mean of the training ratio. This is the middle band of the Bollinger Bands. `center=False` means the average is calculated using past data only (not future data).
   - `std = train.rolling(window=20, center=False).std()`: Calculates the 20-day rolling standard deviation of the training ratio. This measures the volatility of the ratio.
   - `upper_band = movavg + (k * std)`: Calculates the upper Bollinger Band by adding `k` times the standard deviation to the moving average.
   - `lower_band = movavg - (k * std)`: Calculates the lower Bollinger Band by subtracting `k` times the standard deviation from the moving average.
2. **Generate Buy and Sell Signals:**
   - `buy = train.copy()`: Creates a copy of the training ratio to store buy signals.
   - `sell = train.copy()`: Creates a copy of the training ratio to store sell signals.

    - `buy[train > lower_band] = 0`: Sets the `buy` signal to 0 when the training ratio is above the lower Bollinger Band. This means we only have a buy signal when the ratio is *below* the lower band (oversold).
    - `sell[train < upper_band] = 0`: Sets the `sell` signal to 0 when the training ratio is below the upper Bollinger Band. This means we only have a sell signal when the ratio is *above* the upper band (overbought).
3. **Plot Ratio and Trading Signals:**
    - `plt.figure(figsize=(12,7))`: Creates a new figure for the plot.
    - `plt.plot(ratio, label='Ratio')`: Plots the entire ratio (not just the training set) to give context to the buy and sell signals.
    - `buy.plot(color='g', linestyle='None', marker='^')`: Plots the buy signals as green upward-pointing triangles. `linestyle='None'` means the points are plotted without connecting lines.
    - `sell.plot(color='r', linestyle='None', marker='^')`: Plots the sell signals as red upward-pointing triangles.
    - `plt.axis((x1, x2, ratio.min(), ratio.max()))`: Sets the axis limits to the minimum and maximum values of the ratio.
    - `plt.xlim('2010-01-01', '2017-01-01')`: Sets the x-axis limits to the training period.
    - `plt.legend(['Train', 'Buy Signal', 'Sell Signal'])`: Adds a legend to the plot.
    - `plt.show()`: Displays the plot.
4. **Plot Stock Prices and Trading Signals:**
    - `S1 = df['ADBE'].iloc[:1761]`: Extracts the closing prices of ADBE for the training period.
    - `S2 = df['MSFT'].iloc[:1761]`: Extracts the closing prices of MSFT for the training period.
    - `buyR = 0 * S1.copy()`: Creates a copy of `S1` filled with zeros to store buy signals for the stock prices.
    - `sellR = 0 * S1.copy()`: Creates a copy of `S1` filled with zeros to store sell signals for the stock prices.
    - `buyR[buy != 0] = S1[buy != 0]`: When you buy the ratio (meaning you think MSFT is relatively undervalued compared to ADBE), you buy ADBE and sell MSFT. So, this sets the `buyR` signal to the price of ADBE at the buy signal points.
    - `sellR[buy != 0] = S2[buy != 0]`: Similarly, this sets the `sellR` signal to the price of MSFT at the buy signal points (because you're selling MSFT when you buy the ratio).
    - `buyR[sell != 0] = S2[sell != 0]`: When you sell the ratio (meaning you think ADBE is relatively overvalued compared to MSFT), you sell ADBE and buy MSFT. So, this sets the `buyR` signal to the price of MSFT at the sell signal points.

    - `sellR[sell != 0] = S1[sell != 0]`: Similarly, this sets the `sellR` signal to the price of ADBE at the sell signal points (because you're selling ADBE when you sell the ratio).
    - The remaining lines plot the stock prices of ADBE and MSFT, along with the buy and sell signals.

Why It Works:

This code block visualizes the trading strategy by:

- Calculating Bollinger Bands on the training ratio.
- Generating buy and sell signals based on deviations from the bands.
- Plotting the ratio and the trading signals.
- Plotting the stock prices and the corresponding buy/sell signals.

The Bollinger Bands provide a dynamic way to identify overbought and oversold conditions. The plots allow for a visual assessment of the strategy's performance and help to identify potential issues.

Relevance to Project:

This code block provides a visual representation of the trading strategy's logic. The plot of the ratio and the buy/sell signals helps to understand how the strategy generates trading decisions. The plot of the stock prices and the signals shows how the strategy translates these decisions into actual trades. This visualization is useful for debugging, parameter tuning, and communicating the strategy's logic.

Plots:

- **First Plot (Ratio and Signals):** This plot shows the ratio between ADBE and MSFT prices over time. The green upward triangles represent buy signals, which occur when the ratio drops below the lower Bollinger Band, suggesting that ADBE is relatively undervalued compared to MSFT. Conversely, the red upward triangles represent sell signals, triggered when the ratio rises above the upper Bollinger Band, indicating that ADBE is relatively overvalued.
- **Second Plot (Stock Prices and Signals):** This plot visualizes the buy and sell signals in relation to the actual stock prices of ADBE and MSFT. The blue line represents the price of ADBE, and the cyan line represents the price of MSFT. Green upward triangles indicate when to buy ADBE and sell MSFT (when the ratio is low), while red upward triangles indicate when to sell ADBE and buy MSFT (when the ratio is high).

This allows you to visualize when the strategy is buying/selling each individual stock, relative to the relationship between the two.

```
threshold=0.8
```

```python
def trade(S1, S2, window, k, risk_percentage, prev_moni):
    if window == 0 or window >= len(S1):
        return 0, 0  # Return profit and empty trades list

    ratio = S1 / S2

    df1 = pd.DataFrame({'y':S2,'x':S1})
    state_means = 
KalmanFilterRegression(KalmanFilterAverage(df1['x']),KalmanFilterAvera
ge(df1['y']))

    df1['hr'] = - state_means[:,0]
    df1['bee'] = -state_means[:,1]
    df1['spread'] = df1.y + (df1.x * df1['hr']) + (df1['bee'])

    ma = df1.spread.rolling(window=window).mean()
    std = df1.spread.rolling(window=window).std()
    upper_band = ma + (k * std)
    lower_band = ma - (k * std)
    upper_band_soft = ma + (threshold*k*std)
    lower_band_soft = ma - (threshold*k*std)

    money = prev_moni
    countS1 = 0
    countS2 = 0
    trades = []
    daily_moni =[]
    position_size = 1
    for i in range(len(ratio)):
        if np.isnan(ma.iloc[i]) or np.isnan(std.iloc[i]):
            daily_moni.append(0)  # Append current money for days with 
NaN values
            continue

        # Trading logic
        if df1['spread'].iloc[i] > upper_band.iloc[i]:
            money += position_size * (S1.iloc[i] - S2.iloc[i])
            daily_moni.append(position_size * (S1.iloc[i] -
S2.iloc[i]))
            countS1 -= position_size
            countS2 += position_size * ratio.iloc[i]
            position_size=max(1,risk_percentage*money/(S1.iloc[i] +
S2.iloc[i]))

        elif df1['spread'].iloc[i] < lower_band.iloc[i]:
            money -= position_size * (S1.iloc[i] - S2.iloc[i])
            daily_moni.append(-position_size * (S1.iloc[i] -
S2.iloc[i]))
            countS1 += position_size
            countS2 -= position_
```

- **Explanation:** This code imports libraries, downloads historical stock data for ADBE and MSFT from Yahoo Finance (from 2010-01-01 to 2020-01-01), calculates the ratio of their closing prices, and splits the data into training (70%), testing (20%), and out-of-sample (10%) sets. It then plots the training ratio.
- **Why It Works:** `yfinance` enables easy access to historical stock data. The ratio is the core signal for the pairs trade. Splitting the data prevents overfitting and allows for robust evaluation.
- **Relevance to Project:** Provides the foundation for the entire analysis: data and ratio.

Relevance to Project: Informs the choice of parameters like the window size and the lookback period of strategy

4. Spread Calculation and Signal Generation (Bollinger Bands)

python

```
regression_results = KalmanFilterRegression(KalmanFilterAverage(x),
KalmanFilterAverage(y))

a_estimate = regression_results[:, 0]  # Slope (alpha)
b_estimate = regression_results[:, 1]  # Intercept (beta)
```

```python
# Print the last estimates for slope and intercept
print(f"Estimated slope (a): {a_estimate[-1]:.4f}")
print(f"Estimated intercept (b): {b_estimate[-1]:.4f}")

spread = y - (a_estimate + x*b_estimate)
spread_mean =  spread.mean()
spread_std = spread.std()
```

- **Explanation:** Calculates the spread using the time-varying alpha and beta estimates from the Kalman filter regression. Then calculates the mean and standard deviation of the spread
- **Why It Works:** The spread represents the divergence between the two stocks, and its mean and standard deviation are used to define trading signals.
- **Relevance to Project:** Key for generating trading signals based on deviations from the mean. The dynamic alpha and beta from the Kalman filter are central to this.

```python
k=1
movavg=train.rolling(window=20, center= False).mean()

#calculating standard deviation (volatility)
std=train.rolling(window=20, center=False).std()

upper_band= movavg + (k*std)
lower_band= movavg - (k*std)

plt.figure(figsize=(12,7))
plt.plot(ratio, label='Ratio')
buy = train.copy()
sell = train.copy()
buy[train>lower_band] = 0
sell[train<upper_band] = 0

plt.figure(figsize=(12,7))
plt.plot(train, label='ratio')

buy.plot(color='g', linestyle='None', marker='^')
sell.plot(color='r', linestyle='None', marker='^')
x1, x2, y1, y2 = plt.axis()
```

```
plt.axis((x1, x2, ratio.min(), ratio.max()))
plt.xlim('2010-01-01','2017-01-01')
plt.legend(['Train', 'Buy Signal', 'Sell Signal'])
plt.show()

plt.figure(figsize=(12,7))
S1 = df['ADBE'].iloc[:1761]
S2 = df['MSFT'].iloc[:1761]

S1[60:].plot(color='b')
S2[60:].plot(color='c')
buyR = 0*S1.copy()
sellR = 0*S1.copy()

# When you buy the ratio, you buy stock S1 and sell S2
buyR[buy!=0] = S1[buy!=0]
sellR[buy!=0] = S2[buy!=0]

# When you sell the ratio, you sell stock S1 and buy S2
buyR[sell!=0] = S2[sell!=0]
sellR[sell!=0] = S1[sell!=0]

buyR[60:].plot(color='g', linestyle='None', marker='^')
sellR[60:].plot(color='r', linestyle='None', marker='^')
x1, x2, y1, y2 = plt.axis()
plt.axis((x1, x2, min(S1.min(), S2.min()), max(S1.max(), S2.max())))
plt.ylim(25, 105)
plt.xlim('2013-03-22', '2016-07-04')

plt.legend(['ADBE', 'MSFT', 'Buy Signal', 'Sell Signal'])
plt.show()
```

- **Explanation:** This section calculates Bollinger Bands using the rolling mean and standard deviation of the *training ratio*. It then generates buy and sell signals when the ratio crosses the lower and upper bands, respectively. Finally, it plots the ratio, the bands, and the buy/sell signals. It also plots the individual stock prices with the signals overlaid.
- **Why It Works:** Bollinger Bands provide dynamic overbought/oversold levels. Crossing these levels generates signals based on mean reversion.

- **Relevance to Project:** Visualizes the trading strategy and how signals are generated from the ratio. It shows when to buy/sell each stock.

Graph Showing Buy and Sell Signals for All Stocks

The code above generates exactly this. The final plot in that snippet shows:

- The price of ADBE over time (blue line).
- The price of MSFT over time (cyan line).
- Green upward triangles: Points where the strategy buys ADBE and sells MSFT.
- Red upward triangles: Points where the strategy sells ADBE and buys MSFT.

This plot directly visualizes the buy and sell signals for each of the stocks in the pair.

```
final_pnl = cumulative_pnl.iloc[-1]
print(f"Final P&L: {final_pnl}")
```