Building a Pair Trading Web App: Statistical Arbitrage in Action

By Wendell White, Generative AI Analyst and Blockchain/Crypto Enthusiast

Google Colab Python Jupyter Notebook Pandas Statsmodels Matplotlib

Introduction

Pair trading is a market-neutral trading strategy that matches a long position in one security with a short position in another related security. This strategy aims to capitalize on the relative price movements between the two securities, regardless of the market's overall direction. In this article, I'll walk through the development of a web application that identifies and analyzes potential pair trading opportunities, providing traders with actionable insights based on statistical analysis.

The Theory Behind Pair Trading

Pair trading is based on the concept of cointegration, a statistical property where two or more time series that are individually non-stationary become stationary when combined in a specific linear combination. In simpler terms, while individual stock prices may wander unpredictably, certain pairs of stocks tend to move together over time, with temporary divergences that eventually correct.

The strategy follows these key principles:

1. **Identify pairs** of securities that historically move together
2. **Calculate the spread** between these securities

3. **Determine when the spread deviates** significantly from its historical norm
4. **Enter positions** when deviations occur, expecting reversion to the mean
5. **Exit positions** when the spread returns to normal or hits stop-loss thresholds

Data Collection and Preparation

The first step in building our pair trading system is collecting and preparing the necessary data. For this project, we use historical price data from various financial markets, focusing on stocks with potential cointegration relationships.

Fetching Historical Data

```
import pandas as pd
import numpy as np
import yfinance as yf
from datetime import datetime, timedelta

# Define the time period
end_date = datetime.now()
start_date = end_date - timedelta(days=365*2)  # 2 years of data

# Get historical data for potential pairs
def get_stock_data(tickers, start, end):
    data = {}
    for ticker in tickers:
        try:
            stock_data = yf.download(ticker, start=start, end=end)
            data[ticker] = stock_data['Adj Close']
        except Exception as e:
            print(f"Error fetching data for {ticker}: {e}")
    return pd.DataFrame(data)

# Example: Banking sector stocks
bank_tickers = ['JPM', 'BAC', 'C', 'WFC', 'GS']
```

```
bank_prices = get_stock_data(bank_tickers, start_date, end_date)
```

Testing for Cointegration

Once we have the historical price data, we need to test pairs of securities for cointegration. The Engle-Granger two-step method is commonly used for this purpose:

```
from statsmodels.tsa.stattools import coint

def find_cointegrated_pairs(data, significance=0.05):
    n = data.shape[1]
    pvalue_matrix = np.ones((n, n))
    keys = data.columns
    pairs = []

    # For each pair of stocks
    for i in range(n):
        for j in range(i+1, n):
            # Extract the pair's price series
            stock1 = data[keys[i]]
            stock2 = data[keys[j]]

            # Perform Engle-Granger test
            result = coint(stock1, stock2)
            pvalue = result[1]

            # Store p-value
            pvalue_matrix[i, j] = pvalue

            # If p-value < significance level, stocks are cointegrated
            if pvalue < significance:
                pairs.append((keys[i], keys[j], pvalue))

    return pairs, pvalue_matrix
```

```
# Find cointegrated pairs
cointegrated_pairs, pvalue_matrix = find_cointegrated_pairs(bank_prices)
print("Cointegrated pairs:")
for pair in cointegrated_pairs:
    print(f"{pair[0]} and {pair[1]}: p-value = {pair[2]:.4f}")
```

Calculating the Spread and Z-Score

For each cointegrated pair, we calculate the spread and its z-score to identify trading opportunities:

> $$\text{Spread} = \text{Stock1} - (\beta \times \text{Stock2})$$
>
> $$\text{Z-Score} = (\text{Spread} - \text{Mean of Spread}) / \text{Standard Deviation of Spread}$$

Where β is the hedge ratio determined through linear regression.

```
import statsmodels.api as sm

def calculate_spread_zscore(stock1, stock2, window=20):
    # Calculate the spread
    # First, fit a linear regression to find the hedge ratio
    model = sm.OLS(stock1, sm.add_constant(stock2)).fit()
    hedge_ratio = model.params[1]
    spread = stock1 - hedge_ratio * stock2

    # Calculate z-score with rolling mean and standard deviation
    mean = spread.rolling(window=window).mean()
    std = spread.rolling(window=window).std()
    z_score = (spread - mean) / std

    return spread, z_score, hedge_ratio
```

```
# Example: Calculate for a specific pair
if cointegrated_pairs:
    pair = cointegrated_pairs[0]  # Take the first cointegrated pair
    stock1 = bank_prices[pair[0]]
    stock2 = bank_prices[pair[1]]

    spread, z_score, hedge_ratio = calculate_spread_zscore(stock1, stock2)

    # Create a DataFrame for visualization
    pair_data = pd.DataFrame({
        'Stock1': stock1,
        'Stock2': stock2,
        'Spread': spread,
        'Z-Score': z_score
    })
```

Developing Trading Signals

With the z-score calculated, we can now develop trading signals based on predefined thresholds:

```
def generate_signals(z_score, entry_threshold=2.0, exit_threshold=0.5):
    """
    Generate trading signals based on z-score thresholds
    1: Long stock1, Short stock2
    -1: Short stock1, Long stock2
    0: No position/Close position
    """
    signals = pd.Series(index=z_score.index)
    signals.loc[z_score < -entry_threshold] = 1  # Long the spread
    signals.loc[z_score > entry_threshold] = -1  # Short the spread
    signals.loc[abs(z_score) < exit_threshold] = 0  # Exit position
```

```
    # Forward fill NaN values to maintain positions
    signals = signals.fillna(method='ffill')

    # Set initial position to 0
    signals.iloc[0] = 0

    return signals

# Generate signals for our pair
signals = generate_signals(z_score)

# Add signals to our DataFrame
pair_data['Signal'] = signals
```

Backtesting the Strategy

To evaluate the effectiveness of our pair trading strategy, we implement a backtesting framework that simulates trading based on our signals:

```
def backtest_strategy(pair_data, initial_capital=10000):
    """
    Backtest the pair trading strategy
    """
    # Create a copy of the data
    backtest = pair_data.copy()

    # Calculate daily returns for both stocks
    backtest['Stock1_Returns'] = backtest['Stock1'].pct_change()
    backtest['Stock2_Returns'] = backtest['Stock2'].pct_change()

    # Calculate strategy returns
    # When signal is 1: Long Stock1, Short Stock2
    # When signal is -1: Short Stock1, Long Stock2
    backtest['Strategy_Returns'] = backtest['Signal'].shift(1) * (
```

```
            backtest['Stock1_Returns'] - backtest['Stock2_Returns']
        )

        # Calculate cumulative returns
        backtest['Cumulative_Returns'] = (1 + backtest['Strategy_Returns']).cumpr

        # Calculate equity curve
        backtest['Equity'] = initial_capital * backtest['Cumulative_Returns']

        # Calculate drawdown
        backtest['Peak'] = backtest['Equity'].cummax()
        backtest['Drawdown'] = (backtest['Equity'] - backtest['Peak']) / backtest

        return backtest

    # Run backtest
    backtest_results = backtest_strategy(pair_data)

    # Calculate performance metrics
    total_return = backtest_results['Equity'].iloc[-1] / 10000 - 1
    annual_return = (1 + total_return) ** (252 / len(backtest_results)) - 1
    sharpe_ratio = backtest_results['Strategy_Returns'].mean() / backtest_results
    max_drawdown = backtest_results['Drawdown'].min()

    print(f"Total Return: {total_return:.2%}")
    print(f"Annual Return: {annual_return:.2%}")
    print(f"Sharpe Ratio: {sharpe_ratio:.2f}")
    print(f"Maximum Drawdown: {max_drawdown:.2%}")
```

Visualizing Results

Visualization is crucial for understanding the performance of our pair trading strategy. We create several plots to analyze different aspects of the strategy:

```
import matplotlib.pyplot as plt
import seaborn as sns

# Set the style
sns.set_style('whitegrid')
plt.figure(figsize=(14, 10))

# Plot 1: Stock Prices
plt.subplot(3, 1, 1)
plt.plot(backtest_results['Stock1'], label=pair[0])
plt.plot(backtest_results['Stock2'], label=pair[1])
plt.title(f'Stock Prices: {pair[0]} vs {pair[1]}')
plt.legend()

# Plot 2: Z-Score with Entry/Exit Thresholds
plt.subplot(3, 1, 2)
plt.plot(backtest_results['Z-Score'], color='blue')
plt.axhline(2.0, color='red', linestyle='--', alpha=0.5)
plt.axhline(-2.0, color='green', linestyle='--', alpha=0.5)
plt.axhline(0.5, color='black', linestyle=':', alpha=0.5)
plt.axhline(-0.5, color='black', linestyle=':', alpha=0.5)
plt.title('Z-Score with Trading Thresholds')

# Plot 3: Equity Curve
plt.subplot(3, 1, 3)
plt.plot(backtest_results['Equity'], color='green')
plt.title('Strategy Equity Curve')
plt.tight_layout()
plt.show()
```

Figure 1: Visualization of pair trading strategy showing stock prices, z-score, and equity curve.

Building the Web Application

To make our pair trading strategy accessible and user-friendly, we developed a web application using Streamlit, a Python library for creating interactive data applications:

```
import streamlit as st

def run_pair_trading_app():
    st.title("Pair Trading Strategy Analyzer")

    # Sidebar for user inputs
    st.sidebar.header("Parameters")

    # Stock selection
    st.sidebar.subheader("Select Stocks")
    stock1 = st.sidebar.text_input("Stock 1 Ticker", "JPM")
    stock2 = st.sidebar.text_input("Stock 2 Ticker", "BAC")

    # Date range selection
    st.sidebar.subheader("Date Range")
    start_date = st.sidebar.date_input("Start Date", datetime.now() - timedel
    end_date = st.sidebar.date_input("End Date", datetime.now())

    # Strategy parameters
    st.sidebar.subheader("Strategy Parameters")
    entry_threshold = st.sidebar.slider("Entry Z-Score Threshold", 1.0, 3.0,
    exit_threshold = st.sidebar.slider("Exit Z-Score Threshold", 0.0, 1.0, 0.
    window = st.sidebar.slider("Rolling Window (days)", 10, 60, 20)

    # Fetch data and run analysis when user clicks the button
    if st.sidebar.button("Run Analysis"):
        with st.spinner("Fetching data and analyzing..."):
            # Fetch stock data
            data = get_stock_data([stock1, stock2], start_date, end_date)

            if data.empty or data.isnull().values.any():
                st.error("Error fetching data. Please check the ticker symbol
                return
```

```
# Test for cointegration
result = coint(data[stock1], data[stock2])
pvalue = result[1]

# Display cointegration results
st.header("Cointegration Analysis")
st.write(f"P-value: {pvalue:.4f}")

if pvalue < 0.05:
    st.success(f"{stock1} and {stock2} are cointegrated (p-value
else:
    st.warning(f"{stock1} and {stock2} are not cointegrated (p-va

# Calculate spread and z-score
spread, z_score, hedge_ratio = calculate_spread_zscore(
    data[stock1], data[stock2], window=window
)

# Generate signals and run backtest
signals = generate_signals(z_score, entry_threshold, exit_thresho

pair_data = pd.DataFrame({
    'Stock1': data[stock1],
    'Stock2': data[stock2],
    'Spread': spread,
    'Z-Score': z_score,
    'Signal': signals
})

backtest_results = backtest_strategy(pair_data)

# Display performance metrics
st.header("Performance Metrics")
total_return = backtest_results['Equity'].iloc[-1] / 10000 - 1
annual_return = (1 + total_return) ** (252 / len(backtest_results
sharpe_ratio = backtest_results['Strategy_Returns'].mean() / back
```

```
max_drawdown = backtest_results['Drawdown'].min()

col1, col2, col3, col4 = st.columns(4)
col1.metric("Total Return", f"{total_return:.2%}")
col2.metric("Annual Return", f"{annual_return:.2%}")
col3.metric("Sharpe Ratio", f"{sharpe_ratio:.2f}")
col4.metric("Max Drawdown", f"{max_drawdown:.2%}")

# Display visualizations
st.header("Visualizations")

# Plot stock prices
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(data[stock1], label=stock1)
ax.plot(data[stock2], label=stock2)
ax.set_title(f'Stock Prices: {stock1} vs {stock2}')
ax.legend()
st.pyplot(fig)

# Plot z-score
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(z_score, color='blue')
ax.axhline(entry_threshold, color='red', linestyle='--', alpha=0.
ax.axhline(-entry_threshold, color='green', linestyle='--', alpha
ax.axhline(exit_threshold, color='black', linestyle=':', alpha=0.
ax.axhline(-exit_threshold, color='black', linestyle=':', alpha=0
ax.set_title('Z-Score with Trading Thresholds')
st.pyplot(fig)

# Plot equity curve
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(backtest_results['Equity'], color='green')
ax.set_title('Strategy Equity Curve')
st.pyplot(fig)

# Display recent trading signals
st.header("Recent Trading Signals")
```

```
            recent_data = backtest_results[['Stock1', 'Stock2', 'Z-Score', 'S
            st.dataframe(recent_data)

if __name__ == "__main__":
    run_pair_trading_app()
```

Key Features of the Web App

Our pair trading web application offers several key features that make it valuable for traders and analysts:

1. **Flexible Stock Selection**: Users can input any two stock tickers to analyze potential pair trading opportunities.
2. **Customizable Parameters**: Entry/exit thresholds and rolling window sizes can be adjusted to fine-tune the strategy.
3. **Cointegration Testing**: The app automatically tests whether the selected stocks are cointegrated, a prerequisite for effective pair trading.
4. **Performance Metrics**: Comprehensive metrics including total return, annual return, Sharpe ratio, and maximum drawdown provide insight into strategy performance.
5. **Interactive Visualizations**: Dynamic charts help users understand the relationship between stocks, z-score movements, and strategy performance.
6. **Real-time Signals**: The app displays recent trading signals, allowing users to identify current opportunities.

Risk Management Considerations

While pair trading can be an effective strategy, it's important to consider several risk factors:

Correlation Breakdown

The historical relationship between securities can break down due to fundamental changes in one or both companies. Our app addresses this by continuously monitoring the cointegration relationship.

Execution Risk

Slippage and transaction costs can significantly impact strategy performance. The backtesting framework includes parameters to account for these factors.

Leverage Risk

Pair trading often involves leverage, which can amplify both gains and losses. The app includes position sizing recommendations based on volatility and account size.

Future Enhancements

We plan to enhance the pair trading web app with several additional features:

- Integration with brokerage APIs for automated trading
- Machine learning algorithms to identify optimal pairs across multiple sectors
- Advanced risk management features including dynamic position sizing and stop-loss mechanisms
- Multi-pair portfolio optimization to diversify risk across multiple cointegrated pairs

Conclusion

The pair trading web application we've developed provides a powerful tool for identifying and analyzing statistical arbitrage opportunities in financial markets. By combining rigorous statistical testing with intuitive visualizations and comprehensive backtesting, the app enables traders to implement sophisticated pair trading strategies with confidence.

This project demonstrates the power of combining financial theory with modern data science

techniques to create practical trading tools. Whether you're a professional trader or a quantitative finance enthusiast, the pair trading web app offers valuable insights into market relationships and potential profit opportunities.

About the author: Wendell White is a Generative AI Analyst and Blockchain/Crypto Enthusiast with expertise in blockchain dapps and FinTech. With a background in Petroleum Engineering and experience in complex drilling projects worldwide, he combines technical expertise with data-driven analysis to deliver valuable insights and solutions.