**Step 1: Data Exploration and Data Cleaning**

**1.1. Load the Dataset**

Started by importing necessary libraries and loaded CSV file into a Pandas DataFrame.

import pandas as pd

# Load the dataset (update the path as necessary)

df = pd.read\_csv("TRADE\_ROI.csv")

# Display the first few rows to inspect the data

print(df.head())

**1.2. Inspect the Data Structure**

Examine the DataFrame’s shape, columns, and basic info.

# Check the number of rows and columns

print("DataFrame Shape:", df.shape)

# List the column names

print("Columns:", df.columns.tolist())

# Get data types and non-null counts

print(df.info())

# Look at a sample of the Trade\_History column

print(df['Trade\_History'].head())

**1.3. Clean Column Names (if necessary)**

Ensure that column names have no extra spaces and are in a consistent format.

# Remove any leading/trailing spaces in column names

df.columns = df.columns.str.strip()

**1.4. Handle Missing Values**

Check for missing values. Since both columns are critical, decide whether to drop rows with missing values or impute them.

# Check for missing values

print(df.isna().sum())

# If missing values in either column, drop those rows

df.dropna(subset=['Port\_IDs', 'Trade\_History'], inplace=True)

**1.5. Parse the Trade\_History Column**

Since Trade\_History data is in a JSON-like string format (using single quotes), we can use Python’s ast.literal\_eval to safely parse it. This step converts each string into a list of dictionaries.

import ast

def parse\_trade\_history(history\_str):

try:

# Convert the string representation to a Python object

return ast.literal\_eval(history\_str)

except (ValueError, SyntaxError):

return []

# Apply the parser to the Trade\_History column

df['Parsed\_Trade\_History'] = df['Trade\_History'].apply(parse\_trade\_history)

# Inspect the parsed data

print(df['Parsed\_Trade\_History'].head())

**1.6. Expand the Trade History**

If each Parsed\_Trade\_History is a list of trades, we may want to “explode” this list so that each trade becomes its own row. This makes it easier to work with individual trade records.

# Explode the list into separate rows (each row now represents a single trade)

df\_exploded = df.explode('Parsed\_Trade\_History').reset\_index(drop=True)

# Optionally, drop rows where the parsed history is empty

df\_exploded = df\_exploded[df\_exploded['Parsed\_Trade\_History'].notna()]

# Normalize the dictionary so that each key becomes a column

trade\_details = pd.json\_normalize(df\_exploded['Parsed\_Trade\_History'])

# Add back the Port\_ID for each trade record

trade\_details['Port\_ID'] = df\_exploded['Port\_IDs']

# Display a few rows to verify

print(trade\_details.head())

**1.6. Data Type Conversion and Cleaning Trade Details**

Examine and convert data types as needed. For example, if there is a time field (epoch timestamp), convert it to datetime. Also, ensure that numeric fields such as price, quantity, or realizedProfit are properly converted.

# Convert timestamp if available (assuming field is named 'time' and in milliseconds)

if 'time' in trade\_details.columns:

trade\_details['time'] = pd.to\_datetime(trade\_details['time'], unit='ms')

# Convert numeric columns (adjust column names as needed)

for col in ['price', 'quantity', 'realizedProfit']:

if col in trade\_details.columns:

trade\_details[col] = pd.to\_numeric(trade\_details[col], errors='coerce')

# Check for NaN or infinite values in numeric columns

import numpy as np

numeric\_cols = trade\_details.select\_dtypes(include=[np.number]).columns

print("Numeric columns:\n", trade\_details[numeric\_cols].describe())

# Optionally, drop or impute rows with NaN values in essential numeric columns

trade\_details.dropna(subset=['price', 'quantity', 'realizedProfit'], inplace=True)

# Confirm the cleaning results

print(trade\_details.info())

**Step 2: Feature Engineering**

**2.1. Classify Trades**

* **Objective:** Categorize each trade (e.g., long\_open, long\_close, short\_open, short\_close) based on trade attributes.
* **Example Code:**

def classify\_trade(row):

# conditions based on data's schema; for example:

if row['side'] == 'BUY' and row.get('positionSide', 'LONG') == 'LONG':

return 'long\_open'

elif row['side'] == 'SELL' and row.get('positionSide', 'LONG') == 'LONG':

return 'long\_close'

elif row['side'] == 'SELL' and row.get('positionSide', 'SHORT') == 'SHORT':

return 'short\_open'

elif row['side'] == 'BUY' and row.get('positionSide', 'SHORT') == 'SHORT':

return 'short\_close'

else:

return 'unknown'

if 'side' in trade\_details.columns:

trade\_details['PositionAction'] = trade\_details.apply(classify\_trade, axis=1)

print(trade\_details[['Port\_ID', 'time', 'symbol', 'PositionAction']].head())

**2.2. Create Additional Features**

* **Objective:** Compute fields such as cumulative profit/loss over time, daily returns, or any custom metric needed for later ranking.
* **Example Code:**

# For instance, compute cumulative profit for each account:

trade\_details.sort\_values(by='time', inplace=True)

trade\_details['cumulative\_pnl'] = trade\_details.groupby('Port\_ID')['realizedProfit'].cumsum()

**Step 3: Calculate Financial Metrics Per Account**

**3.1. Define Metrics to Calculate**

* **Metrics Include:**
  + **ROI (Return on Investment):** ROI=Total Realized ProfitTotal Investment×100ROI = \frac{\text{Total Realized Profit}}{\text{Total Investment}} \times 100ROI=Total InvestmentTotal Realized Profit​×100
  + **PnL (Profit and Loss):** Sum of realizedProfit
  + **Sharpe Ratio:** Sharpe Ratio=Mean Daily ReturnStd Dev of Daily Return×252\text{Sharpe Ratio} = \frac{\text{Mean Daily Return}}{\text{Std Dev of Daily Return}} \times \sqrt{252}Sharpe Ratio=Std Dev of Daily ReturnMean Daily Return​×252​
  + **MDD (Maximum Drawdown):** Maximum drop from a peak in cumulative PnL
  + **Win Rate:** Percentage of trades with positive realizedProfit
  + **Winning Positions:** Count of trades with positive realizedProfit
  + **Total Positions:** Total trade count

**3.2. Implement a Function to Calculate Metrics**

* **Example Code:**

def calculate\_metrics(group):

metrics = {}

# Calculate total realized profit (PnL)

total\_realized\_profit = group['realizedProfit'].sum()

# Calculate total investment (assuming investment is captured via BUY trade 'quantity')

total\_investment = group.loc[group['side'] == 'BUY', 'quantity'].sum()

# ROI calculation (avoid division by zero)

metrics['ROI (%)'] = (total\_realized\_profit / total\_investment \* 100) if total\_investment != 0 else np.nan

metrics['Total Realized Profit'] = total\_realized\_profit

# Calculate cumulative PnL for Sharpe and MDD calculations

group = group.sort\_values(by='time')

group['cumulative\_pnl'] = group['realizedProfit'].cumsum()

# Calculate daily return percentage changes (here, you might need to adjust if data is not daily)

group['daily\_return'] = group['cumulative\_pnl'].pct\_change().fillna(0)

# Sharpe Ratio (assuming risk-free rate = 0 and 252 trading days/year)

mean\_return = group['daily\_return'].mean()

std\_return = group['daily\_return'].std()

metrics['Sharpe Ratio'] = (mean\_return / std\_return \* np.sqrt(252)) if std\_return != 0 else np.nan

# Maximum Drawdown (MDD)

cumulative\_pnl = group['cumulative\_pnl']

running\_max = cumulative\_pnl.cummax()

drawdown = running\_max - cumulative\_pnl

metrics['Maximum Drawdown'] = drawdown.max()

# Win Rate and trade counts

total\_trades = group.shape[0]

winning\_trades = group[group['realizedProfit'] > 0].shape[0]

metrics['Win Rate (%)'] = (winning\_trades / total\_trades \* 100) if total\_trades > 0 else np.nan

metrics['Winning Positions'] = winning\_trades

metrics['Total Positions'] = total\_trades

return pd.Series(metrics)

# Group by Port\_ID and compute metrics

account\_metrics = trade\_details.groupby('Port\_ID', group\_keys=False).apply(calculate\_metrics)

# Display the computed metrics

print(account\_metrics.head())

**Step 4: Develop the Ranking Algorithm**

**4.1. Choose Metrics for Ranking**

* **Objective:** Decide which metrics to use for ranking (e.g., ROI, Sharpe Ratio, and PnL) and assign weights to each if needed.

**4.2. Normalize and Weight the Metrics**

* **Objective:** Normalize each metric so that they are on a comparable scale (e.g., using min-max scaling or z-scores), then compute a composite score.
* **Example Code:**

# Example: Use ROI, Sharpe Ratio, and Total Realized Profit for ranking

ranking\_metrics = account\_metrics[['ROI (%)', 'Sharpe Ratio', 'Total Realized Profit']].copy()

# Normalize the metrics using min-max scaling

normalized = (ranking\_metrics - ranking\_metrics.min()) / (ranking\_metrics.max() - ranking\_metrics.min())

# Assign weights to each metric (adjust weights as needed)

weights = {

'ROI (%)': 0.4,

'Sharpe Ratio': 0.4,

'Total Realized Profit': 0.2

}

# Compute a composite score

normalized['Composite Score'] = (normalized['ROI (%)'] \* weights['ROI (%)'] +

normalized['Sharpe Ratio'] \* weights['Sharpe Ratio'] +

normalized['Total Realized Profit'] \* weights['Total Realized Profit'])

# Add the composite score back to the account\_metrics DataFrame

account\_metrics['Composite Score'] = normalized['Composite Score']

# Rank the accounts (higher composite score = better ranking)

account\_metrics['Rank'] = account\_metrics['Composite Score'].rank(ascending=False, method='min')

# Sort the DataFrame by rank

account\_metrics.sort\_values('Rank', inplace=True)

print(account\_metrics.head())

**4.3. Extract Top 20 Accounts**

* **Objective:** Once accounts are ranked, filter for the top 20 performers.
* **Example Code:**

top\_20\_accounts = account\_metrics.nsmallest(20, 'Rank')

print("Top 20 Accounts:")

print(top\_20\_accounts)

**Step 5: Documentation and Deliverables**

**5.1. Prepared the Final Code and Notebook**

* **Deliverable:** A Jupyter Notebook that includes all the steps from data loading to the ranking algorithm with detailed comments.

**5.2. Save the Results**

* **Save the Metrics and Rankings:**

# Save the account metrics to a CSV file

account\_metrics.to\_csv("account\_metrics.csv", index=True)

# Save the top 20 ranked accounts to a CSV file

top\_20\_accounts.to\_csv("top\_20\_accounts.csv", index=True)

**5.3. Report**

* **Content of the Report:**
  + **Methodology:** Explain how you cleaned the data, parsed the JSON, computed the metrics, and developed the ranking algorithm.
  + **Findings:** Summarize key insights from the financial metrics and ranking results.
  + **Assumptions:** Document any assumptions (e.g., risk-free rate for Sharpe Ratio, weights for composite score, handling of missing values).
  + **Next Steps:** Suggestions for further analysis or model improvements.

**Summary**

1. **Data Exploration and Data Cleaning:**
   * Load and inspect the dataset.
   * Clean column names, handle missing values, and parse the JSON-like Trade\_History.
   * Expand and normalize trade details.
2. **Feature Engineering:**
   * Create or classify additional features (trade types, cumulative PnL, etc.).
3. **Calculate Financial Metrics:**
   * Group by account (Port\_ID) and calculate metrics such as ROI, PnL, Sharpe Ratio, MDD, Win Rate, and trade counts.
4. **Ranking Algorithm:**
   * Normalize and weight selected metrics.
   * Compute a composite score, rank accounts, and extract the top 20 performers.
5. **Documentation and Deliverables:**
   * Finalized code in a Jupyter Notebook.
   * Export the metrics and top 20 rankings to CSV files.
   * Prepare a comprehensive report detailing methodology, findings, and assumptions.

By following these detailed steps, I’ll have a complete workflow from data ingestion to a final ranked list of top-performing accounts along with full documentation of methodology and insights.

Top of Form

Bottom of Form