GWP3 Yhasreen Oratile Ebenezer

October 27, 2023

0.1 FINANCIAL DATA

Group Work Project 3 | TASK 5 scenario 1 —

1 TASK 5

[ ]:

```
# import libraries
import pandas_datareader as pdr
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import datetime

yf.pdr_override()

# Using code from FRED API: Get US Economic Data using Python

def get_fred_data(param_list, start_date, end_date):
    df = pdr.DataReader(param_list, "fred", start_date, end_date)
    return df.reset_index()
```

[ ]:

```
pip install fredapi
```

```
Collecting fredapi
  Downloading fredapi-0.5.1-py3-none-any.whl (11 kB)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages
(from fredapi) (1.5.3)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.10/dist-packages (from pandas->fredapi) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
packages (from pandas->fredapi) (2023.3.post1)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-
packages (from pandas->fredapi) (1.23.5)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil>=2.8.1->pandas->fredapi) (1.16.0)
Installing collected packages: fredapi
Successfully installed fredapi-0.5.1
```

[ ]:

```
from fredapi import Fred

fred = Fred(api_key='bc7daea6e6a9ab4aa389984bd751c420')
```

https://fred.stlouisfed.org/series/

1.1 Scenerio 1

1.1.1 Assess Market Entry

Delinquency Rate on Credit Card Loans, All Commercial Banks

[ ]:

```
# Delinquency Rate on Credit Card Loans, All Commercial Banks
delinquency_rates = fred.get_series('DRCCLACBS')
delinquency_rates.tail() # quarterly data
```

[ ]:

```
2022-04-01    1.84
2022-07-01    2.08
2022-10-01    2.25
2023-01-01    2.43
2023-04-01    2.77
dtype: float64
```

Credit Utilization Rate

[ ]:

```
# Large Bank Consumer Credit Card Balances: Utilization: Active Accounts Only:␣
  ↪90th Percentile
credit_utilization_rate = fred.get_series('RCCCBACTIVEUTILPCT90')
credit_utilization_rate.tail() # quarterly data
```

[ ]:

```
2022-04-01    91.05
2022-07-01    92.86
2022-10-01    94.24
2023-01-01    93.12
2023-04-01    93.88
dtype: float64
```

Consumer Debt Service Payments as a Percent of Disposable Personal Income

[ ]:

```
# Consumer Debt Service Payments as a Percent of Disposable Personal Income
consumer_debt_income = fred.get_series('CDSP')
consumer_debt_income.tail() # quarterly data
```

[ ]:

```
2022-04-01    5.863954
2022-07-01    5.880974
2022-10-01    5.923596
2023-01-01    5.848341
2023-04-01    5.835505
dtype: float64
```

```
[ ]: # Unemployment rate
     unemployment_rate = fred.get_series('UNRATE')
     unemployment_rate.tail() # monthly data
```

```
[ ]: 2023-05-01    3.7
     2023-06-01    3.6
     2023-07-01    3.5
     2023-08-01    3.8
     2023-09-01    3.8
     dtype: float64
```

```
[ ]: # Inflation rate
     inflation_rate = fred.get_series('FPCPITOTLZGUSA')
     inflation_rate.tail() # yearly data
```

```
[ ]: 2018-01-01    2.442583
     2019-01-01    1.812210
     2020-01-01    1.233584
     2021-01-01    4.697859
     2022-01-01    8.002800
     dtype: float64
```

```
[ ]: # Unemployment rate
     unemployment_rate_last_12 = unemployment_rate.tail(12).mean()

     # Inflation rate
     inflation_rate = inflation_rate.iloc[-1]

     # Delinquency Rate
     delinquency_rate_last_4 = delinquency_rates.tail(4).mean()

     # Credit Utilization
     credit_utilization_rate_last_4 = credit_utilization_rate.tail(4).mean()

     # Consumer Debt Service Payments as a Percent of Disposable Personal Income
     consumer_debt_income_last_4 = consumer_debt_income.tail(4).mean()

     macro_factors = {
         'inflation': inflation_rate,
         'unemployment': unemployment_rate_last_12,
         'delinquency_rate': delinquency_rate_last_4,
         'credit_utilization_rate': credit_utilization_rate_last_4,
         'consumer_debt_income': consumer_debt_income_last_4
     }

     def assess_market(macro_factors):
```

```
    # For simplicity, let's assume we only consider inflation, unemployment␣
↪rate, delinquency rate,
    # credit utilization rate, and consumer debt service payments

    if (macro_factors['inflation'] < 3 and
        macro_factors['unemployment'] < 5 and
        macro_factors['delinquency_rate'] < 5 and
        macro_factors['credit_utilization_rate'] < 30 and
        macro_factors['consumer_debt_income'] < 20):
        return True
    else:
        return False

# Example for market assessment
if assess_market(macro_factors):
    print("Market is suitable for entering the credit card lending business.")
else:
    print("Market conditions are not favorable for credit card lending.")
```

```
Market conditions are not favorable for credit card lending.
```

1.1.2 Calculate credit score

```
[ ]: def calculate_credit_score(credit_history, income, debt_to_income_ratio):

    # For simplicity, let's assume a basic calculation based on credit history␣
↪and income

    score = 0

    # Credit history factor (assuming it is an integer score between 300 to 850)
    score += credit_history

    # Income factor (assuming higher income leads to a higher credit score)
    score += income // 1000  # Assuming each $1000 in income adds 1 point to␣
↪the credit score

    # Debt-to-income ratio factor (assuming a lower ratio leads to a higher␣
↪credit score)
    if debt_to_income_ratio < 0.3:  # Assuming a good debt-to-income ratio is␣
↪below 0.3
        score += 50

    # Other factors (add or subtract points based on other relevant information)

    return score
```

because we do not have a personal customer data available, we will just create a simple hypothetical data of a customer for illustration.

[ ]:

```python
# Example for credit score calculation
credit_history = 750
income = 60000
debt_to_income_ratio = 0.25

credit_score = calculate_credit_score(credit_history, income, debt_to_income_ratio)
print("Credit Score:", credit_score)
```

```
Credit Score: 860
```

[ ]:

```python
#define minimum threshold for loan approval
threshold = 800
```

[ ]:

```python
if credit_score > threshold:
  print("Congratulations! Loan Approved!")
else:
  print("Loan not approved due to low credit score")
```

```
Congratulations! Loan Approved!
```

1.1.3 REFERENCES (DATA)

1. Federal Reserve Bank of St. Louis. (2023). Unemployment Rate. Retrieved from https://fred.stlouisfed.org/series/UNRATE

2. Federal Reserve Bank of St. Louis. (2023). Delinquency Rate on Credit Card Loans, All Commercial Banks. Retrieved from https://fred.stlouisfed.org/series/DRCCLACBS

3. Federal Reserve Bank of St. Louis. (2023). Large Bank Consumer Credit Card Balances: Utilization: Active Accounts Only: 90th Percentile. Retrieved from https://fred.stlouisfed.org/series/RCCCBACTIVEUTILPCT90

4. Federal Reserve Bank of St. Louis. (2023). Consumer Debt Service Payments as a Percent of Disposable Personal Income. Retrieved from https://fred.stlouisfed.org/series/CDSP

5. Federal Reserve Bank of St. Louis. (2023). Inflation, consumer prices for the United States. Retrieved from https://fred.stlouisfed.org/series/FPCPITOTLZGUSA

[ ]:

2 4. AAPL Equity Investment By Yhasreen

2.0.1 4a. Trading base on Market Sentiment with AAPL Investor's Criteria

[ ]:

```
import random
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt

# Collect Data
df = pd.DataFrame({
    'date': pd.date_range(start='2022-01-01', periods=365, freq='D'),
    'review': [random.choice(['positive', 'negative', 'neutral']) for _ in␣
 ↪range(365)],
    'signal': [random.choice(['buy' ,'sell' ,'sell','sell',␣
 ↪'sell','sell','stay']) for _ in range(365)] ,
    'insider_info': [random.choice(['New Product Launch']) for _ in range(365)]␣
 ↪})

data = yf.download('AAPL', start='2022-01-01', end='2023-01-01', progress=False)
data.reset_index(inplace=True)
merged_df = pd.merge(data, df, left_on='Date', right_on='date', how='inner')

# Calculate 14-day moving average
merged_df['14_day_ma'] = merged_df['Close'].rolling(window=14).mean()

# Calculate RSI
delta = merged_df['Close'].diff()
gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
rs = gain / loss
merged_df['RSI'] = 100 - (100 / (1 + rs))

# Calculate Bollinger Bands
merged_df['std_dev'] = merged_df['Close'].rolling(window=14).std()
merged_df['upper_band'] = merged_df['14_day_ma'] + (merged_df['std_dev'] * 2)
merged_df['lower_band'] = merged_df['14_day_ma'] - (merged_df['std_dev'] * 2)

# Confirm signals based on technical indicators
def confirm_signal(row):
    if row['RSI'] < 30 and row['Close'] < row['14_day_ma'] and row['Close'] <␣
 ↪row['lower_band']:
        return 'buy'
    elif row['RSI'] > 70 and row['Close'] > row['14_day_ma'] and row['Close'] >␣
 ↪row['upper_band']:
        return 'sell'
    else:
        return 'stay'

# Function to validate data accuracy (Ethical Consideration 1)
def validate_data_accuracy(data):
    if data.isnull().sum().any():
        raise ValueError("Data contains missing values.")

# Function to handle data privacy (Ethical Consideration 2)
def restrict_sensitive_data(data, authorized):
    if not authorized:
        if "insider_info" in data.columns:
            data.drop("insider_info", axis=1, inplace=True)

# Function to ensure information symmetry among investors (Ethical␣
 ↪Consideration 3)
def ensure_info_symmetry(data, investor_type):
    if investor_type != "institutional":
        if "insider_info" in data.columns:
            raise ValueError("Unauthorized access to insider information.")

merged_df['confirmed_signal'] = merged_df.apply(confirm_signal ,axis=1)
merged_df['signal_mismatch'] = merged_df['signal'] !=␣
 ↪merged_df['confirmed_signal']
filter_df = merged_df.query('signal_mismatch == False')
filtered_df=filter_df[filter_df['confirmed_signal'].isin(['sell'])]

# Investor class
class Investor:
    def __init__(self, name, investor_type, authorized=False):
        self.name = name
        self.investor_type = investor_type
        self.authorized = authorized

    def make_investment_decision(self, data):
        try:
            validate_data_accuracy(data)
            restrict_sensitive_data(data, self.authorized)
            ensure_info_symmetry(data, self.investor_type)

            # Simulated investment decision (Invest if there is at least one␣
 ↪'sell' signal)
            if any(data['confirmed_signal'] == 'sell'):
                print(f"{self.name} decided to invest in AAPL.")
            else:
                print(f"{self.name} decided not to invest in AAPL.")
        except ValueError as e:
            print(f"Error: {e}")
```

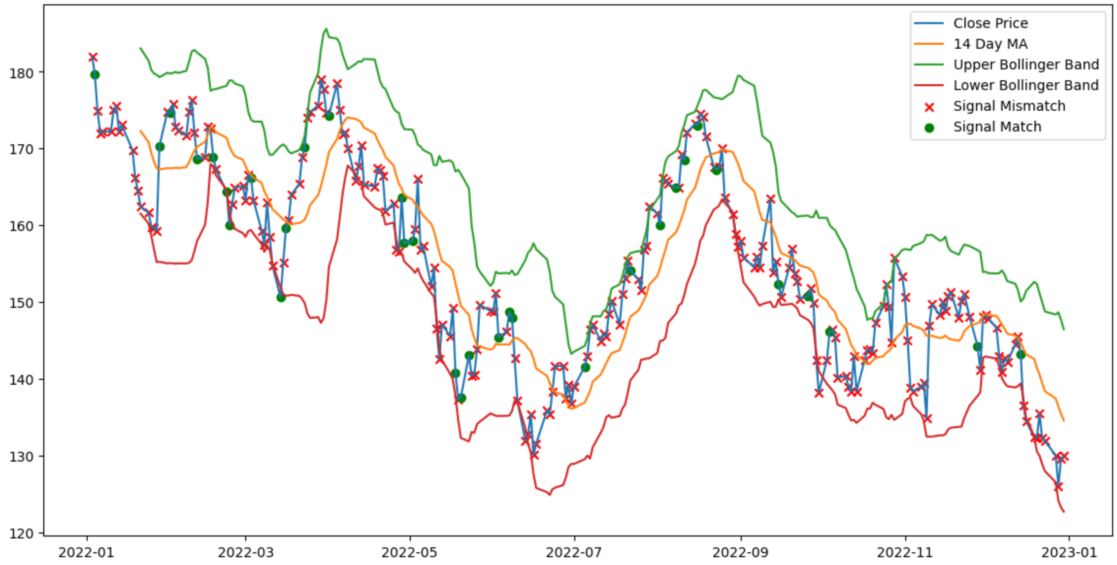
```
# Ethical data considerations
investor1 = Investor("Alice", "individual", False)
investor2 = Investor("Bob Inc.", "institutional", True)

# Making Ethical investment decisions
investor1.make_investment_decision(filtered_df.copy())
investor2.make_investment_decision(filtered_df.copy())
```

```
Alice decided not to invest in AAPL.
Bob Inc. decided not to invest in AAPL.
```

[ ]:

```
# Plot
plt.figure(figsize=(14,7))
plt.plot(merged_df['Date'], merged_df['Close'], label='Close Price')
plt.plot(merged_df['Date'], merged_df['14_day_ma'], label='14 Day MA')
plt.plot(merged_df['Date'], merged_df['upper_band'], label='Upper Bollinger␣
  ↪Band')
plt.plot(merged_df['Date'], merged_df['lower_band'], label='Lower Bollinger␣
  ↪Band')
plt.scatter(merged_df['Date'], merged_df['Close'].
  ↪where(merged_df['signal_mismatch']), label='Signal Mismatch', marker='x',␣
  ↪color='r')
plt.scatter(merged_df['Date'], merged_df['Close'].
  ↪where(~merged_df['signal_mismatch']), label='Signal Match', marker='o',␣
  ↪color='g')
plt.legend(loc='best')
plt.show()
```



[ ]:

```

```

6. Real Estate Portfolio Management System

[ ]:

```python
class Investor:
    def __init__(self, name):
        self.name = name
        self.portfolio = {}

    def lend_security(self, borrower, security, units):
        if security in self.portfolio and self.portfolio[security] >= units:

            if security not in borrower.portfolio:
                borrower.portfolio[security] = 0
            borrower.portfolio[security] += units
            self.portfolio[security] -= units
            print(f"{self.name} lends {units} units of {security} to {borrower.
 ↪name}")
        else:
            print(f"{self.name} cannot lend {units} units of {security} to␣
 ↪{borrower.name}. Insufficient units in the portfolio.")

    def display_portfolio(self):

        print(f"Portfolio of {self.name}:")
        for security, units in self.portfolio.items():
            print(f"{security}: {units} units")

frank = Investor("Frank")
isaac = Investor("Isaac")
sam = Investor("Sam")

frank.portfolio["Pokuase Villa"] = 10
isaac.portfolio["Pokuase Villa"] = 5
sam.portfolio["Pokuase Villa"] = 2

frank.lend_security(isaac, "Pokuase Villa", 3)

frank.display_portfolio()
isaac.display_portfolio()
sam.display_portfolio()
```

```
Frank lends 3 units of Pokuase Villa to Isaac
Portfolio of Frank:
Pokuase Villa: 7 units
```

```
Portfolio of Isaac:
Pokuase Villa: 8 units
Portfolio of Sam:
Pokuase Villa: 2 units
```

[ ]:

```

```