# INTRO

Currently i have the following structure of the project. The goal is to analyze the market volatility for VIB coin and based on the signals and any historical correlations (like volume spikes in some time frame, eth or bitcoin drop, big trade etc etc) predict the outcome for BUY/SELL signals (1% growth/5% growth/10% growth or decline). It stores all the data in SQLITE database. It than send the signals to my telegram bot.

# PROJECT STRUCTURE

1. **Master.py**

#!/usr/bin/env python3

import subprocess

import time

import logging

import os

from logging.handlers import RotatingFileHandler

# Define the base directory for your project

BASE\_DIR = "/Users/igorbulgakov/Documents/vib\_bot"

# Setup logging for the master process with log rotation

MASTER\_LOG\_FILE = os.path.join(BASE\_DIR, "master.log")

logger = logging.getLogger("master")

logger.setLevel(logging.INFO)

rot\_handler = RotatingFileHandler(MASTER\_LOG\_FILE, maxBytes=5\*1024\*1024, backupCount=3)

formatter = logging.Formatter("%(asctime)s [%(levelname)s] %(message)s")

rot\_handler.setFormatter(formatter)

logger.addHandler(rot\_handler)

stream\_handler = logging.StreamHandler()

stream\_handler.setFormatter(formatter)

logger.addHandler(stream\_handler)

logger.info("[MASTER] Starting master.py – Debug log enabled")

print("[MASTER] Starting master.py – Debug log enabled")

# Dictionary of script names and their absolute paths (including vib\_extras)

SCRIPTS = {

"vib\_extras": os.path.join(BASE\_DIR, "vib\_extras.py"),

"vib\_alert": os.path.join(BASE\_DIR, "vib\_alert.py"),

"vib\_master": os.path.join(BASE\_DIR, "vib\_master.py"),

"train\_model\_online\_enhanced": os.path.join(BASE\_DIR, "train\_model\_online\_enhanced.py"),

# "multi\_socket": os.path.join(BASE\_DIR, "multi\_socket.py"),

}

# Corresponding log file paths for each script

LOG\_FILES = {

"vib\_extras": os.path.join(BASE\_DIR, "vib\_extras.log"),

"vib\_alert": os.path.join(BASE\_DIR, "vib\_alert.log"),

"vib\_master": os.path.join(BASE\_DIR, "vib\_master.log"),

"train\_model\_online\_enhanced": os.path.join(BASE\_DIR, "train\_model\_online\_enhanced.log"),

# "multi\_socket": os.path.join(BASE\_DIR, "multi\_socket.log"),

}

# Dictionary to keep track of subprocesses and their log file objects

processes = {}

def start\_script(name, script\_path, log\_path):

"""Starts a script as a subprocess with unbuffered output and logs its output."""

logger.info(f"[MASTER] Starting {name}: {script\_path}")

log\_file = open(log\_path, "a", buffering=1)

proc = subprocess.Popen(

["/usr/bin/python3", "-u", script\_path],

stdout=log\_file,

stderr=log\_file,

cwd=BASE\_DIR

)

processes[name] = (proc, log\_file)

def monitor\_processes():

"""Checks every 30 seconds if any launched process has terminated, and restarts it if needed."""

while True:

for name, (proc, log\_file) in list(processes.items()):

ret = proc.poll()

if ret is not None:

logger.warning(f"[MASTER] {name} terminated (exit code {ret}). Restarting...")

log\_file.close()

start\_script(name, SCRIPTS[name], LOG\_FILES[name])

time.sleep(30)

if \_\_name\_\_ == "\_\_main\_\_":

for name, script\_path in SCRIPTS.items():

start\_script(name, script\_path, LOG\_FILES[name])

monitor\_processes()

1. **multi\_socket.py**

#!/usr/bin/env python3

import threading

import logging

import vib\_alert

import vib\_orderbook

logging.basicConfig(

level=logging.INFO,

format="%(asctime)s [%(levelname)s] %(message)s",

handlers=[logging.StreamHandler()]

)

logger = logging.getLogger("multi\_socket")

def run\_trade\_socket():

try:

logger.info("Starting trade socket (vib\_alert).")

vib\_alert.run\_websocket() # Calls trade feed from vib\_alert.py

except Exception as e:

logger.error("Error in run\_trade\_socket: %s", e)

def run\_orderbook\_socket():

try:

logger.info("Starting orderbook socket (vib\_orderbook).")

vib\_orderbook.run\_orderbook() # Calls order book feed from vib\_orderbook.py

except Exception as e:

logger.error("Error in run\_orderbook\_socket: %s", e)

if \_\_name\_\_ == "\_\_main\_\_":

logger.info("Multi-socket script starting.")

t1 = threading.Thread(target=run\_trade\_socket)

t2 = threading.Thread(target=run\_orderbook\_socket)

t1.start()

t2.start()

t1.join()

t2.join()

1. **train\_model\_online\_enhanced.py**

#!/usr/bin/env python3

import pandas as pd

import numpy as np

import joblib

from datetime import datetime, timedelta

import time

import os

import logging

import sqlite3

import json

from sklearn.metrics import accuracy\_score, classification\_report

# ----------------------------

# Configuration & File Paths

# ----------------------------

BASE\_DIR = "/Users/igorbulgakov/Documents/vib\_bot"

MODEL\_PATH = os.path.join(BASE\_DIR, "model.pkl")

MASTER\_DB\_FILE = os.path.join(BASE\_DIR, "vib\_master.db")

EXTRAS\_DB\_FILE = os.path.join(BASE\_DIR, "vib\_extra\_data.db")

# Feedback window in seconds

FEEDBACK\_WINDOW = 3600

# ----------------------------

# Logging Setup

# ----------------------------

logging.basicConfig(level=logging.INFO,

format="%(asctime)s [%(levelname)s] %(message)s",

handlers=[logging.StreamHandler()])

logger = logging.getLogger(\_\_name\_\_)

# ----------------------------

# DB Functions for Pending Feedback

# ----------------------------

def load\_pending\_feedback():

try:

conn = sqlite3.connect(MASTER\_DB\_FILE)

df = pd.read\_sql\_query("SELECT \* FROM pending\_feedback", conn, parse\_dates=["timestamp"])

conn.close()

return df

except Exception as e:

logger.error(f"Error loading pending feedback: {e}")

return pd.DataFrame()

def delete\_pending\_feedback(record\_id):

try:

conn = sqlite3.connect(MASTER\_DB\_FILE)

cur = conn.cursor()

cur.execute("DELETE FROM pending\_feedback WHERE id = ?", (record\_id,))

conn.commit()

conn.close()

except Exception as e:

logger.error(f"Error deleting pending feedback record {record\_id}: {e}")

def store\_feedback(timestamp, predicted\_label, true\_label):

try:

conn = sqlite3.connect(MASTER\_DB\_FILE)

cur = conn.cursor()

cur.execute("INSERT INTO feedback (timestamp, predicted\_label, true\_label) VALUES (?, ?, ?)",

(timestamp, predicted\_label, true\_label))

conn.commit()

conn.close()

except Exception as e:

logger.error(f"Error storing feedback: {e}")

# ----------------------------

# Utility to Get Latest VIB Price from Extras DB

# ----------------------------

def get\_latest\_vib\_price():

try:

conn = sqlite3.connect(EXTRAS\_DB\_FILE)

df = pd.read\_sql\_query("SELECT \* FROM vib\_extra\_data WHERE symbol='VIBUSDT' ORDER BY close\_time DESC LIMIT 1", conn, parse\_dates=["close\_time"])

conn.close()

if not df.empty:

return df.iloc[0]["close"]

except Exception as e:

logger.error(f"Error fetching latest VIB price: {e}")

return None

# ----------------------------

# Evaluation Function

# ----------------------------

def evaluate\_model\_performance():

"""

Evaluate model performance using feedback records from the master DB.

Logs accuracy and a full classification report.

"""

try:

conn = sqlite3.connect(MASTER\_DB\_FILE)

df\_feedback = pd.read\_sql\_query("SELECT \* FROM feedback", conn, parse\_dates=["timestamp"])

conn.close()

except Exception as e:

logger.error(f"Error loading feedback data for evaluation: {e}")

return

if 'true\_label' in df\_feedback.columns and 'predicted\_label' in df\_feedback.columns and not df\_feedback.empty:

acc = accuracy\_score(df\_feedback['true\_label'], df\_feedback['predicted\_label'])

report = classification\_report(df\_feedback['true\_label'], df\_feedback['predicted\_label'])

logger.info("Post-Update Model Evaluation:")

logger.info(f"Accuracy: {acc:.4f}")

logger.info(f"\n{report}")

else:

logger.error("Feedback data is missing required columns or is empty.")

# ----------------------------

# Training Update Loop

# ----------------------------

def update\_model():

if not os.path.exists(MODEL\_PATH):

logger.error(f"Model not found at {MODEL\_PATH}.")

return

model = joblib.load(MODEL\_PATH)

logger.info("Model loaded for training update.")

df\_pending = load\_pending\_feedback()

if df\_pending.empty:

logger.info("No pending feedback records to process.")

return

current\_time = datetime.now()

latest\_vib\_price = get\_latest\_vib\_price()

if latest\_vib\_price is None:

logger.error("Could not retrieve latest VIB price; aborting update.")

return

updated = False

for idx, row in df\_pending.iterrows():

record\_time = row["timestamp"]

elapsed = (current\_time - record\_time).total\_seconds()

if elapsed < FEEDBACK\_WINDOW:

continue # Skip if feedback window not reached

try:

# Retrieve stored features from JSON

features = np.array([json.loads(row["features"])])

# Assume the stored VIB price (3rd feature) was the prediction price; true feedback is based on the latest VIB price.

predicted\_vib\_price = features[0][2] # Assuming the 3rd feature is the VIB close price at prediction time.

pct\_change = (latest\_vib\_price - predicted\_vib\_price) / predicted\_vib\_price if predicted\_vib\_price else 0.0

# Determine true label based on percentage change

if pct\_change >= 0.10:

true\_label = 3

elif pct\_change >= 0.05:

true\_label = 2

elif pct\_change >= 0.01:

true\_label = 1

elif pct\_change > -0.01:

true\_label = 0

elif pct\_change > -0.05:

true\_label = -1

elif pct\_change > -0.10:

true\_label = -2

else:

true\_label = -3

predicted\_label = row["predicted\_label"]

logger.info(f"Record {row['id']}: Elapsed {elapsed:.2f}s, Predicted {predicted\_label}, True {true\_label}")

# Update model with partial\_fit

model.partial\_fit(features, [true\_label])

# Store feedback in DB

store\_feedback(record\_time.strftime("%Y-%m-%d %H:%M:%S"), predicted\_label, true\_label)

# Delete processed pending feedback

delete\_pending\_feedback(row["id"])

updated = True

except Exception as e:

logger.error(f"Error processing pending feedback record {row['id']}: {e}")

if updated:

try:

joblib.dump(model, MODEL\_PATH)

logger.info("Model updated and saved to disk.")

# Evaluate the model performance after update

evaluate\_model\_performance()

except Exception as e:

logger.error(f"Error saving updated model: {e}")

else:

logger.info("No pending feedback records met the criteria for update.")

def main\_loop():

while True:

update\_model()

# Run update periodically, e.g., every 10 minutes

time.sleep(600)

if \_\_name\_\_ == "\_\_main\_\_":

main\_loop()

1. **vib\_alert.py**

#!/usr/bin/env python3

import websocket

import json

import time

from datetime import datetime

import os

import requests

import logging

import sqlite3

BASE\_DIR = "/Users/igorbulgakov/Documents/vib\_bot"

SOCKET\_URL = "wss://stream.binance.com:9443/ws/vibusdt@trade"

DB\_FILE = os.path.join(BASE\_DIR, "trades.db")

TELEGRAM\_TOKEN = "7636229600:AAESoUoIB6nIcUHxme43x8byKhX1sok5zPk"

CHAT\_ID = 531265494

BIG\_TRADE\_THRESHOLD = 100000

reconnect\_delay = 5

max\_delay = 300

logging.basicConfig(

level=logging.INFO,

format="%(asctime)s [%(levelname)s] %(message)s",

handlers=[logging.StreamHandler()]

)

logger = logging.getLogger("vib\_alert")

def init\_db():

"""Initialize the SQLite database for trades."""

conn = sqlite3.connect(DB\_FILE)

cur = conn.cursor()

cur.execute("""

CREATE TABLE IF NOT EXISTS trades (

id INTEGER PRIMARY KEY AUTOINCREMENT,

local\_time TEXT,

trade\_id INTEGER,

side TEXT,

price REAL,

quantity REAL,

buyer\_order\_id INTEGER,

seller\_order\_id INTEGER,

trade\_time TEXT

);

""")

conn.commit()

conn.close()

logger.info("Trades database initialized and table ensured.")

def insert\_trade(local\_time, trade\_id, side, price, quantity, buyer\_order\_id, seller\_order\_id, trade\_time):

"""Insert a trade record into the trades table."""

try:

conn = sqlite3.connect(DB\_FILE)

cur = conn.cursor()

cur.execute("""

INSERT INTO trades (

local\_time, trade\_id, side, price, quantity, buyer\_order\_id, seller\_order\_id, trade\_time

) VALUES (?, ?, ?, ?, ?, ?, ?, ?)

""", (local\_time, trade\_id, side, price, quantity, buyer\_order\_id, seller\_order\_id, trade\_time))

conn.commit()

except Exception as e:

logger.error(f"Error inserting trade: {e}")

finally:

conn.close()

init\_db()

def send\_telegram\_alert(message, retries=3):

"""Send Telegram alert with retry mechanism."""

url = f"https://api.telegram.org/bot{TELEGRAM\_TOKEN}/sendMessage"

data = {"chat\_id": CHAT\_ID, "text": message}

for attempt in range(retries):

try:

resp = requests.post(url, data=data, timeout=5)

if resp.status\_code == 200:

return

else:

logger.error(f"Telegram Error: {resp.text}")

except Exception as e:

logger.error(f"Telegram Exception on attempt {attempt+1}: {e}")

time.sleep(2)

def on\_message(ws, message):

try:

data = json.loads(message)

except Exception as e:

logger.error(f"Error parsing JSON: {e}")

return

trade\_id = data.get("t")

try:

price = float(data.get("p"))

qty = float(data.get("q"))

except Exception as e:

logger.error(f"Error converting price/qty: {e}")

return

buyer\_order\_id = data.get("b")

seller\_order\_id = data.get("a")

trade\_time\_ms = data.get("T")

is\_buyer\_maker = data.get("m")

local\_time\_str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

try:

trade\_time\_str = datetime.fromtimestamp(trade\_time\_ms / 1000).strftime("%Y-%m-%d %H:%M:%S")

except Exception as e:

logger.error(f"Error converting trade time: {e}")

trade\_time\_str = "N/A"

side = "BUY" if not is\_buyer\_maker else "SELL"

logger.info(f"Trade - ID: {trade\_id} | Side: {side} | Price: {price} | Qty: {qty}")

insert\_trade(local\_time\_str, trade\_id, side, price, qty, buyer\_order\_id, seller\_order\_id, trade\_time\_str)

if qty >= BIG\_TRADE\_THRESHOLD:

total\_value = price \* qty

alert\_text = (

f"🚨 BIG TRADE ALERT 🚨\n"

f"Trade ID: {trade\_id}\n"

f"Side: {side}\n"

f"Price: {price} USDT\n"

f"Amount: {qty} VIB\n"

f"Buyer Order ID: {buyer\_order\_id}\n"

f"Seller Order ID: {seller\_order\_id}\n"

f"Trade Time: {trade\_time\_str}\n"

f"Total Value: {total\_value:.2f} USDT\n"

)

logger.info(alert\_text)

send\_telegram\_alert(alert\_text)

def on\_error(ws, error):

logger.error(f"WebSocket Error: {error}")

def on\_close(ws, close\_status\_code, close\_msg):

global reconnect\_delay

logger.warning("WebSocket Closed.")

logger.warning(f"Reconnecting in {reconnect\_delay} seconds...")

time.sleep(reconnect\_delay)

reconnect\_delay = min(reconnect\_delay \* 2, max\_delay)

run\_websocket()

def on\_open(ws):

global reconnect\_delay

reconnect\_delay = 5

logger.info("Connection Opened... Listening for Big Trades 🧐")

def run\_websocket():

ws = websocket.WebSocketApp(

SOCKET\_URL,

on\_open=on\_open,

on\_message=on\_message,

on\_error=on\_error,

on\_close=on\_close

)

ws.run\_forever(ping\_interval=20, ping\_timeout=10)

if \_\_name\_\_ == "\_\_main\_\_":

while True:

try:

run\_websocket()

except Exception as e:

logger.error(f"Unhandled error: {e}")

logger.info("Retrying in 5 seconds...")

time.sleep(5)

1. **vib\_extras.py**

#!/usr/bin/env python3

import requests

import pandas as pd

import numpy as np

import logging

from datetime import datetime

import time

import os

import sqlite3

import ta # Technical Analysis library

from dotenv import load\_dotenv

# Load environment variables from a .env file if available

load\_dotenv()

# ----------------------------

# Configuration & File Paths

# ----------------------------

BASE\_DIR = os.getenv("BASE\_DIR", "/Users/igorbulgakov/Documents/vib\_bot")

DB\_FILE = os.path.join(BASE\_DIR, "vib\_extra\_data.db") # Database for extra data

# Binance API configuration

API\_URL = os.getenv("BINANCE\_API\_URL", "https://api.binance.com/api/v3/klines")

KLINE\_INTERVAL = os.getenv("KLINE\_INTERVAL", "1m")

KLINE\_LIMIT = int(os.getenv("KLINE\_LIMIT", "500"))

# List of symbols to fetch

SYMBOLS = ["VIBUSDT", "BTCUSDT", "ETHUSDT", "RENDERUSDT"]

# ----------------------------

# Logging Setup

# ----------------------------

logging.basicConfig(

level=logging.INFO,

format="%(asctime)s [%(levelname)s] %(message)s",

handlers=[logging.StreamHandler()]

)

logger = logging.getLogger("vib\_extras")

# ----------------------------

# Database Functions

# ----------------------------

def init\_extra\_data\_db():

"""

Initialize the SQLite database for extra data and create the vib\_extra\_data table if it doesn't exist.

"""

try:

conn = sqlite3.connect(DB\_FILE)

cur = conn.cursor()

cur.execute("""

CREATE TABLE IF NOT EXISTS vib\_extra\_data (

id INTEGER PRIMARY KEY AUTOINCREMENT,

open\_time TEXT,

open REAL,

high REAL,

low REAL,

close REAL,

volume REAL,

close\_time TEXT,

quote\_asset\_volume REAL,

number\_of\_trades INTEGER,

taker\_buy\_base REAL,

taker\_buy\_quote REAL,

ignore TEXT,

rsi REAL,

macd REAL,

macd\_signal REAL,

macd\_hist REAL,

symbol TEXT

);

""")

conn.commit()

logger.info("Database initialized (vib\_extra\_data table ensured).")

except Exception as e:

logger.error(f"Error initializing database: {e}")

finally:

conn.close()

def store\_extra\_data(df):

"""

Store the merged extra data DataFrame into the vib\_extra\_data table.

Parameters:

df (DataFrame): The DataFrame containing extra kline data and computed indicators.

"""

try:

conn = sqlite3.connect(DB\_FILE)

df.to\_sql("vib\_extra\_data", conn, if\_exists="replace", index=False)

conn.commit()

logger.info(f"Stored extra data into database with {len(df)} rows.")

except Exception as e:

logger.error(f"Error storing extra data: {e}")

finally:

conn.close()

# Initialize the database at startup

init\_extra\_data\_db()

# ----------------------------

# Helper Functions for Data Fetching & Indicators

# ----------------------------

def fetch\_klines(symbol, interval, limit=KLINE\_LIMIT):

"""

Fetch klines (candlestick data) for a given symbol from Binance.

Parameters:

symbol (str): The trading pair symbol.

interval (str): The kline interval (e.g., '1m').

limit (int): Maximum number of data points to fetch.

Returns:

DataFrame: A DataFrame with kline data if successful; an empty DataFrame otherwise.

"""

params = {"symbol": symbol, "interval": interval, "limit": limit}

try:

response = requests.get(API\_URL, params=params, timeout=10)

data = response.json()

# If an error occurs, data might be a dict with an error code.

if isinstance(data, dict) and data.get("code"):

logger.error(f"Error fetching klines for {symbol}: {data}")

return pd.DataFrame()

# Define column names per Binance API documentation.

columns = [

"open\_time", "open", "high", "low", "close", "volume",

"close\_time", "quote\_asset\_volume", "number\_of\_trades",

"taker\_buy\_base", "taker\_buy\_quote", "ignore"

]

df = pd.DataFrame(data, columns=columns)

# Convert timestamps from ms to datetime

df["open\_time"] = pd.to\_datetime(df["open\_time"], unit="ms")

df["close\_time"] = pd.to\_datetime(df["close\_time"], unit="ms")

# Convert numeric columns to floats

for col in ["open", "high", "low", "close", "volume",

"quote\_asset\_volume", "taker\_buy\_base", "taker\_buy\_quote"]:

df[col] = pd.to\_numeric(df[col], errors="coerce")

df["symbol"] = symbol

return df

except Exception as e:

logger.error(f"Exception fetching klines for {symbol}: {e}")

return pd.DataFrame()

def compute\_indicators(df):

"""

Compute technical indicators (RSI and MACD) for the given DataFrame and fill NaN values with 0.

Parameters:

df (DataFrame): The DataFrame containing kline data.

Returns:

DataFrame: The input DataFrame with added columns: 'rsi', 'macd', 'macd\_signal', 'macd\_hist'.

"""

try:

df["rsi"] = ta.momentum.rsi(df["close"], window=14).fillna(0)

except Exception as e:

logger.error(f"Error computing RSI for symbol {df.get('symbol', 'unknown')}: {e}")

df["rsi"] = 0

try:

macd = ta.trend.macd(df["close"])

macd\_signal = ta.trend.macd\_signal(df["close"])

df["macd"] = macd.fillna(0)

df["macd\_signal"] = macd\_signal.fillna(0)

df["macd\_hist"] = (macd - macd\_signal).fillna(0)

except Exception as e:

logger.error(f"Error computing MACD for symbol {df.get('symbol', 'unknown')}: {e}")

df["macd"] = 0

df["macd\_signal"] = 0

df["macd\_hist"] = 0

return df

# ----------------------------

# Main Function

# ----------------------------

def main():

"""

Main function to continuously fetch kline data, compute technical indicators,

and store the processed data into the SQLite database.

"""

while True:

all\_data = []

for symbol in SYMBOLS:

logger.info(f"Fetching klines for {symbol}...")

df = fetch\_klines(symbol, KLINE\_INTERVAL, limit=KLINE\_LIMIT)

if df.empty:

logger.warning(f"No data fetched for {symbol}.")

continue

df = compute\_indicators(df)

all\_data.append(df)

time.sleep(0.2) # Pause to avoid hitting API rate limits

if not all\_data:

logger.error("No data fetched for any symbol.")

else:

df\_all = pd.concat(all\_data, ignore\_index=True)

df\_all.sort\_values(by=["symbol", "close\_time"], inplace=True)

logger.info(f"Fetched and computed indicators for {len(df\_all)} rows across all symbols.")

store\_extra\_data(df\_all)

logger.info("Extra data stored in the SQLite database.")

# Wait before the next update cycle (e.g., 60 seconds)

time.sleep(60)

if \_\_name\_\_ == "\_\_main\_\_":

main()

1. **vib\_master.py**

#!/usr/bin/env python3

import pandas as pd

import numpy as np

import joblib

from datetime import datetime, timedelta

import time

import os

import logging

import sqlite3

import json

import requests

# ----------------------------

# Configuration & File Paths

# ----------------------------

BASE\_DIR = "/Users/igorbulgakov/Documents/vib\_bot"

TRADES\_DB\_FILE = os.path.join(BASE\_DIR, "trades.db")

EXTRAS\_DB\_FILE = os.path.join(BASE\_DIR, "vib\_extra\_data.db")

ORDERBOOK\_DB\_FILE = os.path.join(BASE\_DIR, "orderbook.db")

# Model path (trained model to be used for inference)

MODEL\_PATH = os.path.join(BASE\_DIR, "model.pkl")

# Master DB for storing predictions and pending feedback (for training updates)

MASTER\_DB\_FILE = os.path.join(BASE\_DIR, "vib\_master.db")

TELEGRAM\_TOKEN = "7636229600:AAESoUoIB6nIcUHxme43x8byKhX1sok5zPk"

CHAT\_ID = 531265494

# Feedback window (in seconds)

FEEDBACK\_WINDOW = 3600

# ----------------------------

# Logging Setup

# ----------------------------

logger = logging.getLogger("vib\_master")

logger.setLevel(logging.DEBUG)

file\_handler = logging.FileHandler(os.path.join(BASE\_DIR, "vib\_master.log"))

file\_handler.setLevel(logging.DEBUG)

formatter = logging.Formatter("%(asctime)s [%(levelname)s] %(message)s")

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.INFO)

console\_handler.setFormatter(formatter)

logger.addHandler(console\_handler)

# ----------------------------

# DB Write Functions for Master DB

# ----------------------------

def store\_prediction(timestamp, predicted\_label):

conn = sqlite3.connect(MASTER\_DB\_FILE)

cur = conn.cursor()

cur.execute("INSERT INTO predictions (timestamp, predicted\_label) VALUES (?, ?)",

(timestamp, predicted\_label))

conn.commit()

conn.close()

def store\_pending\_feedback(timestamp, predicted\_label, features, vib\_price):

# Save features as JSON

features\_json = json.dumps(features.tolist())

conn = sqlite3.connect(MASTER\_DB\_FILE)

cur = conn.cursor()

cur.execute("INSERT INTO pending\_feedback (timestamp, predicted\_label, features) VALUES (?, ?, ?)",

(timestamp, predicted\_label, features\_json))

conn.commit()

conn.close()

def store\_alert(message):

# Optional: log alert messages or perform further actions.

logger.info(f"Telegram Alert Sent: {message}")

def send\_telegram\_alert(message):

url = f"https://api.telegram.org/bot{TELEGRAM\_TOKEN}/sendMessage"

data = {"chat\_id": CHAT\_ID, "text": message}

try:

resp = requests.post(url, data=data, timeout=5)

if resp.status\_code != 200:

logger.error(f"Telegram Error: {resp.text}")

except Exception as e:

logger.error(f"Telegram Exception: {e}")

# ----------------------------

# Data Loading Functions

# ----------------------------

def load\_trades\_data():

try:

conn = sqlite3.connect(TRADES\_DB\_FILE)

df = pd.read\_sql\_query("SELECT \* FROM trades", conn)

conn.close()

if "trade\_time" in df.columns:

df["trade\_time"] = pd.to\_datetime(df["trade\_time"], errors="coerce")

if "local\_time" in df.columns:

df["local\_time"] = pd.to\_datetime(df["local\_time"], errors="coerce")

logger.info(f"Loaded trades data: {len(df)} rows.")

except Exception as e:

logger.error(f"Error loading trades data: {e}")

df = pd.DataFrame()

return df

def load\_extras\_data():

try:

conn = sqlite3.connect(EXTRAS\_DB\_FILE)

df = pd.read\_sql\_query("SELECT \* FROM vib\_extra\_data", conn, parse\_dates=["open\_time", "close\_time"])

conn.close()

logger.info(f"Loaded extras data: {len(df)} rows.")

except Exception as e:

logger.error(f"Error loading extras data: {e}")

df = pd.DataFrame()

return df

def load\_orderbook\_data():

try:

conn = sqlite3.connect(ORDERBOOK\_DB\_FILE)

df = pd.read\_sql\_query("SELECT \* FROM orderbook\_data", conn, parse\_dates=["timestamp"])

conn.close()

df.sort\_values("timestamp", inplace=True)

logger.info(f"Loaded orderbook data: {len(df)} rows.")

except Exception as e:

logger.error(f"Error loading orderbook data: {e}")

df = pd.DataFrame()

return df

def load\_data():

df\_extras = load\_extras\_data()

df\_trades = load\_trades\_data()

df\_orderbook = load\_orderbook\_data()

if not df\_extras.empty:

df\_extras.sort\_values("open\_time", inplace=True)

return df\_extras, df\_trades, df\_orderbook

# ----------------------------

# Data Merging Function

# ----------------------------

def merge\_data(df\_extras, df\_trades, df\_orderbook):

# Filter VIBUSDT data

df\_vib = df\_extras[df\_extras["symbol"] == "VIBUSDT"].copy()

if df\_vib.empty:

logger.error("No VIBUSDT data found.")

return None

df\_vib.sort\_values("close\_time", inplace=True)

latest\_candle = df\_vib.iloc[-1]

# Count big trades (last 5 minutes)

BIG\_TRADE\_THRESHOLD = 100000

big\_trades\_count = len(df\_trades[

(df\_trades["trade\_time"] >= latest\_candle["close\_time"] - timedelta(minutes=5)) &

(df\_trades["quantity"] >= BIG\_TRADE\_THRESHOLD)

])

# Get orderbook spread

if not df\_orderbook.empty:

valid\_snapshots = df\_orderbook[df\_orderbook["timestamp"] <= latest\_candle["close\_time"]]

orderbook\_spread = valid\_snapshots.iloc[-1]["spread"] if not valid\_snapshots.empty else 0.0

else:

orderbook\_spread = 0.0

# Pivot table for related symbols

other\_symbols = ["BTCUSDT", "ETHUSDT", "RENDERUSDT"]

df\_others = df\_extras[df\_extras["symbol"].isin(other\_symbols)].copy()

if df\_others.empty:

diff\_BTC, diff\_ETH, diff\_RNDR = 0, 0, 0

else:

df\_pivot = df\_others.pivot\_table(index="close\_time", columns="symbol", values="close", aggfunc="last")

df\_pivot.reset\_index(inplace=True)

df\_merged = pd.merge\_asof(df\_vib, df\_pivot, on="close\_time", direction="backward", tolerance=pd.Timedelta(seconds=60))

diff\_BTC = (df\_merged.iloc[-1].get("BTCUSDT", np.nan) - latest\_candle.get("close", 0)) / latest\_candle.get("close", 1)

diff\_ETH = (df\_merged.iloc[-1].get("ETHUSDT", np.nan) - latest\_candle.get("close", 0)) / latest\_candle.get("close", 1)

diff\_RNDR = (df\_merged.iloc[-1].get("RENDERUSDT", np.nan) - latest\_candle.get("close", 0)) / latest\_candle.get("close", 1)

diff\_BTC = 0 if np.isnan(diff\_BTC) else diff\_BTC

diff\_ETH = 0 if np.isnan(diff\_ETH) else diff\_ETH

diff\_RNDR = 0 if np.isnan(diff\_RNDR) else diff\_RNDR

features = [

latest\_candle.get("rsi", 0),

latest\_candle.get("macd\_hist", 0),

latest\_candle.get("close", 0),

latest\_candle.get("volume", 0),

big\_trades\_count,

orderbook\_spread,

diff\_BTC,

diff\_ETH,

diff\_RNDR

]

# Return: features (numpy array), current VIB price, and the candle's timestamp

return np.array([features]), latest\_candle["close"], latest\_candle["close\_time"]

# ----------------------------

# Inference Loop

# ----------------------------

def run\_inference(model):

df\_extras, df\_trades, df\_orderbook = load\_data()

merged = merge\_data(df\_extras, df\_trades, df\_orderbook)

if merged is None:

logger.warning("[ML] Merge failed. Skipping inference cycle.")

return

X, current\_vib\_price, current\_timestamp = merged

try:

prediction = model.predict(X)[0]

except Exception as e:

logger.error(f"[ML] Prediction error: {e}")

return

timestamp\_str = current\_timestamp.strftime("%Y-%m-%d %H:%M:%S")

store\_prediction(timestamp\_str, prediction)

logger.info(f"[ML] Prediction: {prediction}")

# Store pending feedback record for later training update

store\_pending\_feedback(timestamp\_str, prediction, X, current\_vib\_price)

if prediction != 0:

alert\_msg = f"Signal: {prediction}\nFeatures: {X[0].tolist()}"

send\_telegram\_alert(alert\_msg)

def main\_loop():

if not os.path.exists(MODEL\_PATH):

logger.error(f"Model not found at {MODEL\_PATH}.")

return

model = joblib.load(MODEL\_PATH)

logger.info("Model loaded successfully for inference.")

while True:

run\_inference(model)

time.sleep(30)

if \_\_name\_\_ == "\_\_main\_\_":

main\_loop()

1. **vib\_orderbook.py**

#!/usr/bin/env python3

import websocket

import json

import time

from datetime import datetime

import os

import sqlite3

import logging

BASE\_DIR = "/Users/igorbulgakov/Documents/vib\_bot"

ORDER\_BOOK\_URL = "wss://stream.binance.com:9443/ws/vibusdt@depth5"

DB\_FILE = os.path.join(BASE\_DIR, "orderbook.db")

logging.basicConfig(

level=logging.INFO,

format="%(asctime)s [%(levelname)s] %(message)s",

handlers=[logging.StreamHandler()]

)

logger = logging.getLogger("vib\_orderbook")

def init\_db():

"""Initialize the SQLite database for orderbook snapshots."""

conn = sqlite3.connect(DB\_FILE)

cur = conn.cursor()

cur.execute("""

CREATE TABLE IF NOT EXISTS orderbook\_data (

id INTEGER PRIMARY KEY AUTOINCREMENT,

timestamp TEXT,

best\_bid REAL,

best\_ask REAL,

spread REAL

);

""")

conn.commit()

conn.close()

logger.info("Orderbook database initialized (orderbook\_data table ensured).")

init\_db()

def insert\_orderbook(best\_bid, best\_ask, spread):

"""Insert an orderbook snapshot into the database."""

try:

conn = sqlite3.connect(DB\_FILE)

cur = conn.cursor()

timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

cur.execute("""

INSERT INTO orderbook\_data (timestamp, best\_bid, best\_ask, spread)

VALUES (?, ?, ?, ?)

""", (timestamp, best\_bid, best\_ask, spread))

conn.commit()

except Exception as e:

logger.error(f"Error inserting orderbook snapshot: {e}")

finally:

conn.close()

def on\_message(ws, message):

try:

data = json.loads(message)

except Exception as e:

logger.error(f"Error parsing JSON: {e}")

return

bids = data.get('bids', [])

asks = data.get('asks', [])

logger.info(f"Received orderbook snapshot: {len(bids)} bids, {len(asks)} asks")

if bids and asks:

try:

best\_bid = round(float(bids[0][0]), 4)

best\_ask = round(float(asks[0][0]), 4)

spread = best\_ask - best\_bid

logger.info(f"Orderbook: Bid {best\_bid}, Ask {best\_ask}, Spread {spread:.4f}")

insert\_orderbook(best\_bid, best\_ask, spread)

except Exception as e:

logger.error(f"Error processing orderbook data: {e}")

def on\_error(ws, error):

logger.error(f"WebSocket Error: {error}")

def on\_close(ws, close\_status\_code, close\_msg):

logger.warning("Orderbook WebSocket Closed. Reconnecting in 5 seconds...")

time.sleep(5)

run\_orderbook()

def on\_open(ws):

logger.info("Orderbook WebSocket connected. Listening for updates...")

def run\_orderbook():

ws = websocket.WebSocketApp(

ORDER\_BOOK\_URL,

on\_open=on\_open,

on\_message=on\_message,

on\_error=on\_error,

on\_close=on\_close

)

ws.run\_forever(ping\_interval=20, ping\_timeout=10)

if \_\_name\_\_ == "\_\_main\_\_":

while True:

try:

run\_orderbook()

except Exception as e:

logger.error(f"Unhandled error: {e}")

logger.info("Retrying in 5 seconds...")

time.sleep(5)