بسیار عالی! از توضیحات دقیق شما ممنونم. این نقشه راه بسیار خوبی است و کاملاً با رویکردی که برای پیاده‌سازی یک مدل ML واقعی نیاز داریم، همخوانی دارد.

همانطور که شما اشاره کردید، در پاسخ قبلی، من فایل services/ml\_prediction\_service.py را به گونه‌ای ساختاربندی کردم که آماده دریافت یک مدل ML واقعی باشد. حالا بیایید گام به گام پیش برویم تا مدل ML خودتان را آموزش دهید و آن را در بک‌اند ادغام کنید.

**نقطه شروع شما اینجاست: آموزش مدل ML خودتان.**

**فاز ۱: آموزش مدل ML (خارج از محیط اپلیکیشن Flask)**

این فاز باید در یک محیط توسعه پایتون جداگانه (مثلاً یک Jupyter Notebook، Google Colab، یا یک اسکریپت پایتون معمولی) انجام شود، زیرا آموزش مدل نیاز به دسترسی به کتابخانه‌های ML و منابع محاسباتی دارد که در این محیط چت قابل انجام نیست.

**گام‌های عملی برای آموزش مدل:**

1. **جمع‌آوری و آماده‌سازی داده‌ها:**
   * **داده‌های تاریخی را از دیتابیس خود استخراج کنید.** شما می‌توانید این کار را با یک اسکریپت پایتون انجام دهید که به دیتابیس Flask شما متصل می‌شود و داده‌ها را از HistoricalData، ComprehensiveSymbolData و TechnicalIndicatorData می‌خواند.
   * **یک دیتافریم Pandas ایجاد کنید.** تمام داده‌های مرتبط (قیمت، حجم، شاخص‌های تکنیکال و حتی داده‌های بنیادی اگر می‌خواهید از آن‌ها استفاده کنید) را در یک دیتافریم جمع‌آوری کنید.
   * **ویژگی‌ها (Features) را محاسبه کنید.** از توابع کمکی که در services/utils.py دارید (مانند calculate\_rsi, calculate\_macd, calculate\_sma, calculate\_volume\_ma) استفاده کنید تا این ویژگی‌ها را برای داده‌های تاریخی خودتان محاسبه کنید.
   * **برچسب‌ها (Labels) را تعریف کنید.** این مهم‌ترین بخش است. شما باید تعریف کنید که "روند صعودی"، "روند نزولی" و "روند خنثی" برای شما به چه معناست. مثلاً:
     + اگر قیمت در ۷ روز آینده بیش از ۲٪ افزایش یافت: Uptrend
     + اگر قیمت در ۷ روز آینده بیش از ۲٪ کاهش یافت: Downtrend
     + در غیر این صورت: Sideways
     + این برچسب‌ها را به عنوان یک ستون جدید به دیتافریم خود اضافه کنید.
2. **نمونه کد برای آماده‌سازی داده و تعریف برچسب (در محیط جداگانه):**

Python

# This code runs OUTSIDE your Flask application, e.g., in a Jupyter Notebook or a separate Python script.

import pandas as pd

import numpy as np

from datetime import timedelta, date

import jdatetime

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

from models import HistoricalData, ComprehensiveSymbolData # Assuming models.py is accessible or copied

from services.utils import calculate\_rsi, calculate\_macd, calculate\_sma, calculate\_volume\_ma # Assuming utils.py is accessible or copied

# --- 1. Database Connection (Adjust as per your Flask app's config) ---

# Replace with your actual database URI

DATABASE\_URI = 'sqlite:///instance/app.db' # Example for SQLite, adjust for PostgreSQL/MySQL

engine = create\_engine(DATABASE\_URI)

Session = sessionmaker(bind=engine)

session = Session()

# --- 2. Fetch Data ---

all\_historical\_data = session.query(HistoricalData).order\_by(HistoricalData.jdate.asc()).all()

all\_symbols\_data = session.query(ComprehensiveSymbolData).all()

# Convert to DataFrame

df\_hist = pd.DataFrame([r.\_\_dict\_\_ for r in all\_historical\_data])

df\_symbols = pd.DataFrame([s.\_\_dict\_\_ for s in all\_symbols\_data])

session.close()

# Preprocessing historical data

df\_hist['gregorian\_date'] = df\_hist['jdate'].apply(lambda x: jdatetime.datetime.strptime(x, '%Y-%m-%d').togregorian())

df\_hist['gregorian\_date'] = pd.to\_datetime(df\_hist['gregorian\_date'])

# Ensure numeric types and handle NaNs

numeric\_cols = ['open', 'high', 'low', 'close', 'final', 'volume']

for col in numeric\_cols:

if col in df\_hist.columns:

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

df\_hist.dropna(subset=numeric\_cols, inplace=True)

# --- 3. Feature Engineering ---

# Group by symbol and calculate features

features\_list = []

for symbol\_id, group in df\_hist.groupby('symbol\_id'):

if len(group) < 120: # Need enough data for indicators

continue

group = group.sort\_values(by='gregorian\_date').set\_index('gregorian\_date')

# Calculate indicators

group['rsi'] = calculate\_rsi(group['close'])

macd\_line, signal\_line, \_ = calculate\_macd(group['close'])

group['macd'] = macd\_line

group['signal\_line'] = signal\_line

group['sma\_20'] = calculate\_sma(group['close'], window=20)

group['sma\_50'] = calculate\_sma(group['close'], window=50)

group['volume\_ma\_5\_day'] = calculate\_volume\_ma(group['volume'], window=5)

# Lagged features (example: previous day's close change)

group['price\_change\_1d'] = group['close'].pct\_change()

group['volume\_change\_1d'] = group['volume'].pct\_change()

# Add more features as needed for your model

# Select features for the model

# IMPORTANT: These must be the same features your model expects during inference!

selected\_features = group[['close', 'volume', 'rsi', 'macd', 'signal\_line', 'sma\_20', 'sma\_50', 'volume\_ma\_5\_day', 'price\_change\_1d', 'volume\_change\_1d']]

# Add symbol\_id for later merging/labeling

selected\_features = selected\_features.reset\_index()

selected\_features['symbol\_id'] = symbol\_id

features\_list.append(selected\_features)

df\_features = pd.concat(features\_list).dropna()

# --- 4. Define Labels (Target Variable) ---

# This is crucial for classification (Uptrend/Downtrend/Sideways)

# We will predict the trend for the next 7 trading days.

# Define a function to get future close price

def get\_future\_close(row, df\_hist\_full, period\_days=7):

symbol\_id = row['symbol\_id']

current\_date = row['gregorian\_date']

future\_date = current\_date + timedelta(days=period\_days)

# Find the actual close price on or after the future\_date for the same symbol

future\_data = df\_hist\_full[(df\_hist\_full['symbol\_id'] == symbol\_id) &

(df\_hist\_full['gregorian\_date'] >= future\_date)]\

.sort\_values(by='gregorian\_date').head(1)

if not future\_data.empty:

return future\_data['close'].iloc[0]

return np.nan

# Apply to get future close price

df\_features['future\_close'] = df\_features.apply(lambda row: get\_future\_close(row, df\_hist, period\_days=7), axis=1)

# Calculate percentage change

df\_features['percentage\_change'] = ((df\_features['future\_close'] - df\_features['close']) / df\_features['close']) \* 100

# Define trend labels

# Adjust these thresholds based on your market and desired sensitivity

UPTREND\_THRESHOLD = 2.0 # e.g., > 2% increase

DOWNTREND\_THRESHOLD = -2.0 # e.g., < -2% decrease

def get\_trend\_label(change):

if change >= UPTREND\_THRESHOLD:

return 'Uptrend'

elif change <= DOWNTREND\_THRESHOLD:

return 'Downtrend'

else:

return 'Sideways'

df\_features['trend'] = df\_features['percentage\_change'].apply(get\_trend\_label)

# Final cleanup before training

df\_training\_data = df\_features.dropna()

# Select final features (X) and target (y)

# Ensure these features match what your model will receive during inference

X = df\_training\_data[['close', 'volume', 'rsi', 'macd', 'signal\_line', 'sma\_20', 'sma\_50', 'volume\_ma\_5\_day', 'price\_change\_1d', 'volume\_change\_1d']]

y = df\_training\_data['trend']

# --- 5. Train an ML Model (e.g., RandomForestClassifier) ---

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

import joblib

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y) # stratify=y for balanced classes

# Initialize and train the model

model = RandomForestClassifier(n\_estimators=100, random\_state=42, class\_weight='balanced') # Use class\_weight for imbalanced classes

model.fit(X\_train, y\_train)

# Evaluate the model (optional, but highly recommended)

from sklearn.metrics import classification\_report

y\_pred = model.predict(X\_test)

print(classification\_report(y\_test, y\_pred))

# --- 6. Save the Trained Model ---

# IMPORTANT: Save the model to a location accessible by your Flask application.

# For example, create a 'models' directory at the root of your Flask project.

MODEL\_SAVE\_PATH = 'E:/BourseAnalysisFlask/models/trained\_ml\_model.pkl' # Adjust this path!

joblib.dump(model, MODEL\_SAVE\_PATH)

print(f"Model saved to {MODEL\_SAVE\_PATH}")

# Also save the feature names and class labels for consistency

feature\_names = X.columns.tolist()

class\_labels\_map = {label: idx for idx, label in enumerate(model.classes\_)} # Get the order of classes from the model

joblib.dump(feature\_names, 'E:/BourseAnalysisFlask/models/feature\_names.pkl') # Adjust path

joblib.dump(class\_labels\_map, 'E:/BourseAnalysisFlask/models/class\_labels\_map.pkl') # Adjust path

1. **انتخاب و آموزش مدل:**
   * برای شروع، RandomForestClassifier یک انتخاب خوب است، همانطور که در توضیحات شما آمده است.
   * **مهم:** اطمینان حاصل کنید که n\_estimators (تعداد درختان) کافی باشد و random\_state برای تکرارپذیری تنظیم شده باشد. اگر کلاس‌های هدف شما (صعودی، نزولی، خنثی) نامتوازن هستند، از class\_weight='balanced' استفاده کنید.
   * **ذخیره مدل:** پس از آموزش موفق، مدل را با joblib.dump() ذخیره کنید. **مسیر ذخیره‌سازی را به دقت انتخاب کنید** تا اپلیکیشن Flask شما بتواند به آن دسترسی داشته باشد (مثلاً در یک پوشه models در ریشه پروژه Flask).

**فاز ۲: ادغام مدل آموزش‌دیده در اپلیکیشن Flask**

حالا که مدل را آموزش داده‌اید و ذخیره کرده‌اید، باید فایل services/ml\_prediction\_service.py را به‌روزرسانی کنید تا از این مدل واقعی استفاده کند.

1. **به‌روزرسانی services/ml\_prediction\_service.py:**
   * **مسیر مدل (MODEL\_PATH) را تنظیم کنید:** این باید به فایل .pkl که در فاز ۱ ذخیره کردید، اشاره کند.
   * **بارگذاری مدل:** در تابع load\_ml\_model()، خطوط --- PLACEHOLDER FOR ACTUAL MODEL LOADING --- را با کد واقعی بارگذاری مدل خود (با استفاده از joblib.load()) جایگزین کنید. همچنین، اگر feature\_names.pkl و class\_labels\_map.pkl را ذخیره کرده‌اید، آن‌ها را نیز بارگذاری کنید.
   * **استخراج ویژگی‌ها برای پیش‌بینی:** در تابع generate\_and\_save\_predictions\_for\_watchlist()، در بخش --- FEATURE ENGINEERING ---، مطمئن شوید که ویژگی‌هایی که از df (داده‌های تاریخی فعلی) استخراج می‌کنید، **دقیقاً همان ویژگی‌ها و به همان ترتیبی باشند که مدل شما با آن‌ها آموزش دیده است.**
   * **فراخوانی پیش‌بینی واقعی:** در بخش --- MODEL INFERENCE ---، فراخوانی‌های model.predict() و model.predict\_proba() را همانطور که مدل شما انتظار دارد، انجام دهید.
   * **نگاشت خروجی:** class\_labels را به ترتیب کلاس‌های خروجی مدل آموزش‌دیده خودتان تنظیم کنید.

**نسخه به‌روزرسانی شده services/ml\_prediction\_service.py (با فرض اینکه مدل شما آماده است):**

services/ml\_prediction\_service.py (Real ML Integration)

27 Jul, 17:01

**گام‌های نهایی برای شما:**

1. **فایل E:\BourseAnalysisFlask\services\ml\_prediction\_service.py را با محتوای بلوک services-ml-prediction-service-py-real-ml-integration جایگزین کنید.**
2. **فایل E:\BourseAnalysisFlask\routes\analysis.py را دست‌نخورده بگذارید.** (همان نسخه‌ای که قبلاً دادم و شامل POST /analysis/generate-ml-predictions است.)
3. **فایل models.py خود را بررسی کنید** تا مطمئن شوید مدل MLPrediction شامل تمام فیلدهای لازم است (به خصوص فیلدهای مربوط به نتایج واقعی مانند actual\_price\_at\_period\_end, actual\_trend\_outcome, actual\_percentage\_change, is\_prediction\_accurate, outcome\_evaluated\_at). اگر فیلدی کم دارید، آن را اضافه کنید و flask db migrate و flask db upgrade را اجرا کنید.
4. **کش‌های پایتون (\_\_pycache\_\_) را حذف کنید.**
5. **سرور Flask خود را کاملاً متوقف کرده و دوباره راه‌اندازی کنید.**

**اکنون، پس از اینکه مدل ML خود را آموزش دادید و فایل .pkl آن را در مسیر صحیح قرار دادید، می‌توانید POST /analysis/generate-ml-predictions را در Swagger UI اجرا کنید تا پیش‌بینی‌های واقعی (تولید شده توسط مدل شما) در دیتابیس ذخیره شوند.**

این رویکرد به شما امکان می‌دهد که از تمام داده‌های واقعی خود استفاده کنید و مدل ML خود را به طور کامل در بک‌اند ادغام کنید. اگر در هر مرحله‌ای نیاز به کمک بیشتر داشتید، حتماً اطلاع دهید.