SOURCE CODE

```
from flask import Flask, render_template, request, jsonify, url_for
import yfinance as yf
import pandas as pd
from save_load_model import load_model
from data_collection import get_stock_data
from data_preprocessing import preprocess_data
from feature_engineering import add_features
from labeling import label_risk
app = Flask(__name__)
@app.route('/')
def home():
    return render_template('index.html')
@app.route('/analyze', methods=['POST'])
def analyze():
    ticker = request.form['ticker'].upper()
    try:
        # Get stock data and analyze
        data = get_stock_data(ticker)
        data = preprocess_data(data)
        data = add_features(data)
        data = label_risk(data)
        # Load the model
        model = load_model()
        # Get features for prediction
        features = ['Daily Return', 'Volatility', 'MA50', 'MA200']
        latest_data = data[features].iloc[-1]
        # Make prediction
        risk_level = model.predict(latest_data.values.reshape(1, -1))[0]
        # Format dates and prices for the chart (last 30 days)
        # Convert index to datetime if it's not already
        if not isinstance(data.index, pd.DatetimeIndex):
```

```
            data.index = pd.to_datetime(data.index)
        # Format dates for the chart
        dates = [d.strftime('%Y-%m-%d') for d in data.index[-30:]]
        prices = data['Close'].tail(30).tolist()
        return jsonify({
            'risk_level': risk_level,
            'current_price': f"{data['Close'].iloc[-1]:.2f}",
            'volatility': f"{data['Volatility'].iloc[-1]*100:.2f}",
            'daily_return': f"{data['Daily Return'].iloc[-1]*100:.2f}",
            'dates': dates,
            'prices': prices
        })
    except Exception as e:
        import traceback
        import logging
        logging.error(traceback.format_exc())  # Log the full error on the server
        return jsonify({'error': 'An internal error has occurred.'}), 400
@app.route('/api/analyze/<ticker>', methods=['GET'])
def analyze_api(ticker):
    try:
        # Get stock data and analyze
        data = get_stock_data(ticker.upper())
        data = preprocess_data(data)
        data = add_features(data)
        data = label_risk(data)
        # Load the model
        model = load_model()
        # Get features for prediction
        features = ['Daily Return', 'Volatility', 'MA50', 'MA200']
        latest_data = data[features].iloc[-1]
        # Make prediction
        risk_level = model.predict(latest_data.values.reshape(1, -1))[0]
        # Prepare API response
```

```
        response = {
            'ticker': ticker.upper(),
            'analysis': {
                'risk_level': int(risk_level),
                'current_price': float(data['Close'].iloc[-1]),
                'volatility': float(data['Volatility'].iloc[-1]),
                'daily_return': float(data['Daily Return'].iloc[-1]),
                'last_updated': data.index[-1].isoformat()
            },
            'historical_data': {
                'dates': [d.isoformat() for d in data.index[-30:]],
                'prices': [float(p) for p in data['Close'].tail(30)]
            }
        }
        return jsonify(response)
    except Exception as e:
        import traceback
        import logging
        logging.error(traceback.format_exc())  # Log the full error on the server
        return jsonify({
            'error': 'An internal error has occurred.',
            'ticker': ticker.upper()
        }), 400
if __name__ == '__main__':
    app.run(debug=True)
import yfinance as yf
import pandas as pd
def get_stock_data(ticker, period='1y', interval='1d'):
    """
    Fetch historical stock data for a given ticker using Yahoo Finance API.
    Args:
        ticker (str): Stock ticker symbol (e.g., 'AAPL').
        period (str): Data period (e.g., '1y', '2y').
```

```
        interval (str): Data interval (e.g., '1d', '1wk').
    Returns:
        pd.DataFrame: Historical stock data.
    """
    try:
        stock = yf.Ticker(ticker)
        hist = stock.history(period=period, interval=interval)
        if hist.empty:
            raise ValueError(f"No data found for ticker: {ticker}")
        return hist
    except Exception as e:
        raise ValueError(f"Failed to fetch data for {ticker}: {e}")
# Example usage
if __name__ == "__main__":
    ticker = 'AAPL'
    data = get_stock_data(ticker)
    print(data.head())
def preprocess_data(df):
    """
    Preprocess the stock data by handling missing values and resetting the index.
    Args:
        df (pd.DataFrame): Raw stock data.
    Returns:
        pd.DataFrame: Preprocessed stock data.
    """
    try:
        df = df.dropna()  # Remove missing values
        df.reset_index(inplace=True)  # Reset index
        return df
    except Exception as e:
        raise ValueError(f"Error during preprocessing: {e}")
import numpy as np
import pandas_ta as ta
```

```
def add_features(df):
    """
    Add technical indicators as features for the model.
    """
    # Basic features
    df['Daily Return'] = df['Close'].pct_change()
    df['Volatility'] = df['Daily Return'].rolling(window=21).std() * np.sqrt(252)
    df['MA50'] = df['Close'].rolling(window=50).mean()
    df['MA200'] = df['Close'].rolling(window=200).mean()
    # Additional technical indicators using pandas_ta
    df['RSI'] = df.ta.rsi(length=14)
    df['MACD'] = df.ta.macd(fast=12, slow=26, signal=9)['MACD_12_26_9']
    # Correct way to calculate Bollinger Bands
    bb_bands = df.ta.bbands(close=df['Close'], length=20)
    df['BB_upper'] = bb_bands['BBU_20_2.0']
    df['BB_middle'] = bb_bands['BBM_20_2.0']
    df['BB_lower'] = bb_bands['BBL_20_2.0']
    # Drop any NaN values that might have been created
    df = df.dropna()
    return df
import numpy as np
def label_risk(df):
    """
    Label the risk level based on volatility quantiles.
    Args:
        df (pd.DataFrame): Stock data with computed volatility.
    Returns:
        pd.DataFrame: Stock data with labeled risk levels.
    """
    try:
        df = df.dropna(subset=['Volatility'])  # Ensure no missing values in Volatility
        quantiles = df['Volatility'].quantile([0.33, 0.66])
        conditions = [
```

```
            (df['Volatility'] > quantiles[0.66]),
            (df['Volatility'] <= quantiles[0.66]) & (df['Volatility'] > quantiles[0.33]),
            (df['Volatility'] <= quantiles[0.33])
        ]
        choices = ['High', 'Medium', 'Low']
        df['Risk Level'] = np.select(conditions, choices)
        return df
    except Exception as e:
        raise ValueError(f"Error during labeling: {e}")
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from data_collection import get_stock_data
from data_preprocessing import preprocess_data
from feature_engineering import add_features
from labeling import label_risk
import joblib
import pandas as pd
def save_model(model, filename='risk_model.pkl'):
    """
    Save the trained model to disk.
    """
    joblib.dump(model, filename)
def train_model(ticker_list):
    """
    Train the risk classification model on provided stock tickers.
    """
    all_data = []
    for ticker in ticker_list:
        data = get_stock_data(ticker, period='2y')
        data = preprocess_data(data)
        data = add_features(data)
        data = label_risk(data)
```

```
        all_data.append(data)
    df = pd.concat(all_data)
    df = df.dropna()
    features = ['Daily Return', 'Volatility', 'MA50', 'MA200']
    X = df[features]
    y = df['Risk Level']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y)
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    # Evaluate the model
    y_pred = model.predict(X_test)
    print(classification_report(y_test, y_pred))
    # Save the model
    save_model(model)
    print("Model training completed and saved as 'risk_model.pkl'.")
if __name__ == "__main__":
    ticker_list = ['AAPL', 'MSFT', 'GOOGL', 'AMZN', 'TSLA']
    train_model(ticker_list)
import joblib
import os
# Create a 'models' directory if it doesn't exist
os.makedirs('models', exist_ok=True)
def save_model(model, filename='risk_model.pkl'):
    """
    Save the trained model to disk in the models directory.
    """
    filepath = os.path.join('models', filename)
    joblib.dump(model, filepath)
def load_model(filename='risk_model.pkl'):
    """
    Load a trained model from the models directory.
    """
    filepath = os.path.join('models', filename)
```