Stock Price Prediction System

This notebook implements an advanced stock price prediction system using machine learning models including XGBoost ensemble and LSTM neural networks.

Import Libraries

```
!pip install yfinance xgboost tensorflow scikit-learn pandas numpy
matplotlib plotly seaborn --quiet

import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
import warnings
warnings.filterwarnings('ignore')

from sklearn.preprocessing import MinMaxScaler, StandardScaler,
RobustScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score
from sklearn.model_selection import train_test_split, TimeSeriesSplit
import xgboost as xgb
from sklearn.ensemble import RandomForestRegressor,
GradientBoostingRegressor
from sklearn.linear_model import Ridge, ElasticNet

try:
    import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import LSTM, Dense, Dropout,
BatchNormalization, GRU
    from tensorflow.keras.optimizers import Adam
    from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau
    LSTM_AVAILABLE = True
except ImportError:
    print("TensorFlow not available. LSTM model will be skipped.")
    LSTM_AVAILABLE = False

from datetime import datetime, timedelta
```

Data Fetching Functions

```
def get_valid_ticker():
    while True:
        stock = input("Enter a valid stock ticker (e.g., AAPL, TSLA, 
MSFT): ").upper()
        try:
            test = yf.Ticker(stock)
            if test.history(period="1d").empty:
                print("Invalid ticker. Please try again.")
            else:
                return stock
        except Exception as e:
            print(f"Error validating ticker: {e}")
            print("Invalid input. Please try again.")

def get_model_choice():
    print("\nChoose prediction model:")
    print("1. XGBoost (Less time, High accuracy)")
    print("2. LSTM (More time, Most accuracy)")
    print("3. Both (Conclusion with both models)")

    while True:
        choice = input("Enter your choice (1/2/3): ")
        if choice in ['1', '2', '3']:
            return int(choice)
        print("Invalid choice. Please enter 1, 2, or 3.")

def fetch_stock_data(ticker_symbol, period="max"):
    ticker = yf.Ticker(ticker_symbol)
    data = ticker.history(period=period)

    if data.empty:
        print(f"No data found for {ticker_symbol}")
        return None

    info = ticker.info
    company_name = info.get('longName', ticker_symbol)

    print(f"\nFetched data for {company_name} ({ticker_symbol})")
    print(f"Data range: {data.index[0].date()} to {data.index[-
1].date()}")
    print(f"Total trading days: {len(data)}")

    return data, company_name
```

Technical Indicators

```
def calculate_rsi(prices, window=14):
    delta = prices.diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=window).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=window).mean()
    rs = gain / loss
    rsi = 100 - (100 / (1 + rs))
    return rsi

def calculate_macd(prices, fast=12, slow=26, signal=9):
    ema_fast = prices.ewm(span=fast).mean()
    ema_slow = prices.ewm(span=slow).mean()
    macd = ema_fast - ema_slow
    macd_signal = macd.ewm(span=signal).mean()
    macd_hist = macd - macd_signal
    return macd, macd_signal, macd_hist

def calculate_bollinger_bands(prices, window=20, num_std=2):
    rolling_mean = prices.rolling(window=window).mean()
    rolling_std = prices.rolling(window=window).std()
    bb_upper = rolling_mean + (rolling_std * num_std)
    bb_lower = rolling_mean - (rolling_std * num_std)
    return bb_upper, rolling_mean, bb_lower

def calculate_stochastic(high, low, close, k_window=14, d_window=3):
    lowest_low = low.rolling(window=k_window).min()
    highest_high = high.rolling(window=k_window).max()
    stoch_k = 100 * ((close - lowest_low) / (highest_high -
lowest_low))
    stoch_d = stoch_k.rolling(window=d_window).mean()
    return stoch_k, stoch_d

def calculate_atr(high, low, close, window=14):
    tr1 = high - low
    tr2 = abs(high - close.shift())
    tr3 = abs(low - close.shift())
    true_range = pd.concat([tr1, tr2, tr3], axis=1).max(axis=1)
    atr = true_range.rolling(window=window).mean()
    return atr

def calculate_williams_r(high, low, close, window=14):
    highest_high = high.rolling(window=window).max()
    lowest_low = low.rolling(window=window).min()
    williams_r = -100 * ((highest_high - close) / (highest_high -
lowest_low))
    return williams_r

def add_technical_indicators(data):
    df = data.copy()

    df['Returns'] = df['Close'].pct_change()
    df['Log_Returns'] = np.log(df['Close'] / df['Close'].shift(1))
    df['High_Low_Pct'] = (df['High'] - df['Low']) / df['Close']
    df['Open_Close_Pct'] = (df['Close'] - df['Open']) / df['Open']
    df['Price_Volume'] = df['Close'] * df['Volume']
    df['Volume_Rate'] = df['Volume'] /
df['Volume'].rolling(window=20).mean()

    for period in [3, 5, 10, 20, 50, 100]:
        df[f'MA_{period}'] = df['Close'].rolling(window=period).mean()
        df[f'MA_{period}_ratio'] = df['Close'] / df[f'MA_{period}']
        df[f'EMA_{period}'] = df['Close'].ewm(span=period).mean()
        df[f'EMA_{period}_ratio'] = df['Close'] / df[f'EMA_{period}']
        df[f'Close_MA_{period}_diff'] = df['Close'] -
df[f'MA_{period}']

    df['RSI'] = calculate_rsi(df['Close'])
    df['RSI_7'] = calculate_rsi(df['Close'], 7)
    df['RSI_21'] = calculate_rsi(df['Close'], 21)

    df['MACD'], df['MACD_Signal'], df['MACD_Hist'] =
calculate_macd(df['Close'])

    df['BB_Upper'], df['BB_Middle'], df['BB_Lower'] =
calculate_bollinger_bands(df['Close'])
    df['BB_Width'] = (df['BB_Upper'] - df['BB_Lower']) /
df['BB_Middle']
    df['BB_Position'] = (df['Close'] - df['BB_Lower']) /
(df['BB_Upper'] - df['BB_Lower'])

    df['Stoch_K'], df['Stoch_D'] = calculate_stochastic(df['High'],
df['Low'], df['Close'])
    df['ATR'] = calculate_atr(df['High'], df['Low'], df['Close'])
    df['Williams_R'] = calculate_williams_r(df['High'], df['Low'],
df['Close'])

    for period in [5, 10, 20]:
        df[f'Vol_MA_{period}'] =
df['Volume'].rolling(window=period).mean()
        df[f'Vol_Std_{period}'] =
df['Volume'].rolling(window=period).std()
        df[f'Price_Volatility_{period}'] =
df['Close'].rolling(window=period).std()
        df[f'High_MA_{period}'] =
df['High'].rolling(window=period).mean()
        df[f'Low_MA_{period}'] =
df['Low'].rolling(window=period).mean()

    return df
```

Feature Engineering

```
def create_sequence_features(data, target_col='Close',
sequence_length=10):
    df = data.copy()

    price_cols = ['Close', 'High', 'Low', 'Open', 'Volume']
    indicator_cols = ['RSI', 'MACD', 'BB_Position', 'Stoch_K', 'ATR']

    for col in price_cols + indicator_cols:
        if col in df.columns:
            for i in range(1, sequence_length + 1):
                df[f'{col}_lag_{i}'] = df[col].shift(i)

    for window in [3, 5, 10]:
        df[f'{target_col}_momentum_{window}'] = df[target_col] /
df[target_col].shift(window) - 1
        df[f'{target_col}_volatility_{window}'] =
df[target_col].rolling(window).std() /
df[target_col].rolling(window).mean()

    return df

def prepare_features_xgboost(data, look_back=20, forecast_horizon=1):
    df = data.copy()
    df = create_sequence_features(df, sequence_length=look_back)

    for i in range(1, forecast_horizon + 1):
        df[f'Target_{i}'] = df['Close'].shift(-i)

    df = df.dropna()
    return df
```

Model Training Functions

```
# XGBoost ensemble model implementation

def train_xgboost_model(data, forecast_horizon=1):
    print("\nTraining Enhanced XGBoost model...")

    target_cols = [f'Target_{i}' for i in range(1, forecast_horizon +
1)]
    feature_cols = [col for col in data.columns if col not in
target_cols + ['Open', 'Close', 'High', 'Low', 'Volume', 'Adj Close']]

    X = data[feature_cols]
    y = data[target_cols[0]] if len(target_cols) == 1 else
data[target_cols]

    X = X.select_dtypes(include=[np.number])
    X = X.fillna(X.mean())

    split_ratio = 0.85
    split_idx = int(len(X) * split_ratio)
    X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
    y_train, y_test = y.iloc[:split_idx], y.iloc[split_idx:]

    scaler = RobustScaler()
    X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train),
columns=X_train.columns, index=X_train.index)
    X_test_scaled = pd.DataFrame(scaler.transform(X_test),
columns=X_test.columns, index=X_test.index)

    models = []

    xgb_model = xgb.XGBRegressor(
        n_estimators=300,
        max_depth=6,
        learning_rate=0.03,
        subsample=0.9,
        colsample_bytree=0.9,
        reg_alpha=0.05,
        reg_lambda=0.05,
        random_state=42,
        n_jobs=-1
    )

    rf_model = RandomForestRegressor(
        n_estimators=200,
        max_depth=10,
        min_samples_split=5,
        min_samples_leaf=2,
        random_state=42,
        n_jobs=-1
    )

    gb_model = GradientBoostingRegressor(
        n_estimators=200,
        max_depth=6,
        learning_rate=0.05,
        subsample=0.9,
        random_state=42
    )

    models = [('XGB', xgb_model), ('RF', rf_model), ('GB', gb_model)]

    ensemble_train_pred = np.zeros(len(y_train))
    ensemble_test_pred = np.zeros(len(y_test))

    for name, model in models:
        model.fit(X_train_scaled, y_train)
        train_pred = model.predict(X_train_scaled)
        test_pred = model.predict(X_test_scaled)

        ensemble_train_pred += train_pred / len(models)
        ensemble_test_pred += test_pred / len(models)

    train_rmse = np.sqrt(mean_squared_error(y_train,
ensemble_train_pred))
    test_rmse = np.sqrt(mean_squared_error(y_test,
ensemble_test_pred))
    train_r2 = r2_score(y_train, ensemble_train_pred)
    test_r2 = r2_score(y_test, ensemble_test_pred)

    print(f"Ensemble Training RMSE: {train_rmse:.4f}")
    print(f"Ensemble Testing RMSE: {test_rmse:.4f}")
    print(f"Ensemble Training R²: {train_r2:.4f}")
    print(f"Ensemble Testing R²: {test_r2:.4f}")

    return models, X_test_scaled, y_test.values, ensemble_test_pred,
split_idx, scaler

# LSTM model implementation

def prepare_lstm_data(data, look_back=60, forecast_horizon=1):
    features = ['Close', 'Volume', 'High', 'Low', 'Open']
    if 'RSI' in data.columns:
        features.append('RSI')
    if 'MACD' in data.columns:
        features.append('MACD')
    if 'BB_Position' in data.columns:
        features.append('BB_Position')

    feature_data =
data[features].fillna(method='ffill').fillna(method='bfill')

    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(feature_data)

    X, y = [], []
    for i in range(look_back, len(scaled_data) - forecast_horizon +
1):
        X.append(scaled_data[i-look_back:i])
        y.append(scaled_data[i:i+forecast_horizon, 0])

    X, y = np.array(X), np.array(y)
    if forecast_horizon == 1:
        y = y.reshape(-1)

    return X, y, scaler
```

```python
def train_lstm_model(data, forecast_horizon=1):
    if not LSTM_AVAILABLE:
        print("LSTM model not available. Skipping...")
        return None, None, None, None, None, None

    print("\nTraining Enhanced LSTM model...")

    X, y, scaler = prepare_lstm_data(data, look_back=60,
forecast_horizon=forecast_horizon)

    split_ratio = 0.85
    split_idx = int(len(X) * split_ratio)
    X_train, X_test = X[:split_idx], X[split_idx:]
    y_train, y_test = y[:split_idx], y[split_idx:]

    model = Sequential([
        LSTM(128, return_sequences=True,
input_shape=(X_train.shape[1], X_train.shape[2])),
        Dropout(0.2),
        BatchNormalization(),
        LSTM(64, return_sequences=True),
        Dropout(0.2),
        BatchNormalization(),
        GRU(32, return_sequences=False),
        Dropout(0.2),
        Dense(64, activation='relu'),
        BatchNormalization(),
        Dropout(0.1),
        Dense(32, activation='relu'),
        Dense(forecast_horizon if forecast_horizon > 1 else 1)
    ])

    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss='huber',
        metrics=['mae']
    )

    early_stopping = EarlyStopping(monitor='val_loss', patience=20,
restore_best_weights=True)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.7,
patience=10, min_lr=0.00001)

    history = model.fit(
        X_train, y_train,
        batch_size=32,
        epochs=150,
        validation_data=(X_test, y_test),
        callbacks=[early_stopping, reduce_lr],
        verbose=0
    )

    train_pred = model.predict(X_train, verbose=0)
    test_pred = model.predict(X_test, verbose=0)

    if forecast_horizon == 1:
        train_pred = train_pred.flatten()
        test_pred = test_pred.flatten()

    train_pred_prices = scaler.inverse_transform(
        np.column_stack([train_pred, np.zeros((len(train_pred),
scaler.n_features_in_ - 1))])
    )[:, 0]
    test_pred_prices = scaler.inverse_transform(
        np.column_stack([test_pred, np.zeros((len(test_pred),
scaler.n_features_in_ - 1))])
    )[:, 0]
    y_train_prices = scaler.inverse_transform(
        np.column_stack([y_train, np.zeros((len(y_train),
scaler.n_features_in_ - 1))])
    )[:, 0]
    y_test_prices = scaler.inverse_transform(
        np.column_stack([y_test, np.zeros((len(y_test),
scaler.n_features_in_ - 1))])
    )[:, 0]

    train_rmse = np.sqrt(mean_squared_error(y_train_prices,
train_pred_prices))
    test_rmse = np.sqrt(mean_squared_error(y_test_prices,
test_pred_prices))
    train_r2 = r2_score(y_train_prices, train_pred_prices)
    test_r2 = r2_score(y_test_prices, test_pred_prices)

    print(f"LSTM Training RMSE: {train_rmse:.4f}")
    print(f"LSTM Testing RMSE: {test_rmse:.4f}")
    print(f"LSTM Training R²: {train_r2:.4f}")
    print(f"LSTM Testing R²: {test_r2:.4f}")

    return model, X_test, y_test_prices, test_pred_prices, split_idx +
60, scaler
```

Trading signals generation and recommendation system

```
def generate_trading_signals(original_data, predictions, start_idx):
    df = original_data.copy()
    df['Predicted'] = np.nan
```

```python
    df['Buy_Signal'] = False
    df['Sell_Signal'] = False

    if len(predictions) > 0:
        end_idx = start_idx + len(predictions)
        if end_idx <= len(df):
            df.iloc[start_idx:end_idx,
df.columns.get_loc('Predicted')] = predictions

    if 'BB_Upper' not in df.columns:
        df['BB_Upper'] = df['Close'].rolling(window=20).mean() +
(df['Close'].rolling(window=20).std() * 2)
        df['BB_Lower'] = df['Close'].rolling(window=20).mean() -
(df['Close'].rolling(window=20).std() * 2)
        df['BB_Middle'] = df['Close'].rolling(window=20).mean()

    df['Signal'] = 0

    for i in range(max(1, start_idx), len(df)):
        if pd.notna(df.iloc[i]['Predicted']):
            current_price = df.iloc[i]['Close']
            predicted_price = df.iloc[i]['Predicted']
            bb_lower = df.iloc[i]['BB_Lower']
            bb_upper = df.iloc[i]['BB_Upper']
            rsi = df.iloc[i].get('RSI', 50)

            price_change = (predicted_price - current_price) /
current_price

            if (current_price <= bb_lower * 1.01 and
                predicted_price > current_price * 1.002 and
                rsi < 45 and price_change > 0.005):
                df.iloc[i, df.columns.get_loc('Buy_Signal')] = True
                df.iloc[i, df.columns.get_loc('Signal')] = 1
            elif (current_price >= bb_upper * 0.99 and
                  predicted_price < current_price * 0.998 and
                  rsi > 55 and price_change < -0.005):
                df.iloc[i, df.columns.get_loc('Sell_Signal')] = True
                df.iloc[i, df.columns.get_loc('Signal')] = -1

    return df

def calculate_recommendation(data, current_price, predicted_price,
rsi, bb_position):
    signals = []

    price_change = (predicted_price - current_price) / current_price
    if price_change > 0.02:
        signals.append(('buy', 0.4))
    elif price_change < -0.02:
        signals.append(('sell', 0.4))
    else:
        signals.append(('hold', 0.3))

    if rsi < 30:
        signals.append(('buy', 0.3))
    elif rsi > 70:
        signals.append(('sell', 0.3))
    else:
        signals.append(('hold', 0.2))

    if bb_position < 0.2:
        signals.append(('buy', 0.25))
    elif bb_position > 0.8:
        signals.append(('sell', 0.25))
    else:
        signals.append(('hold', 0.2))

    buy_prob = sum([weight for action, weight in signals if action == 
'buy'])
    sell_prob = sum([weight for action, weight in signals if action == 
'sell'])
    hold_prob = sum([weight for action, weight in signals if action == 
'hold'])

    total = buy_prob + sell_prob + hold_prob
    buy_prob = (buy_prob / total) * 100
    sell_prob = (sell_prob / total) * 100
    hold_prob = (hold_prob / total) * 100

    return buy_prob, hold_prob, sell_prob
```

Visualization Functions

```
def create_interactive_plot(data_with_signals, company_name, 
ticker_symbol):
    fig = make_subplots(
        rows=2, cols=1,
        subplot_titles=[f'{company_name} ({ticker_symbol}) - Price 
Prediction & Trading Signals', 'Volume'],
        vertical_spacing=0.1,
        row_heights=[0.7, 0.3]
    )

    fig.add_trace(
        go.Scatter(x=data_with_signals.index, 
y=data_with_signals['Close'],
                  name='Actual Price', line=dict(color='blue', 
width=2)),
        row=1, col=1
    )

    predicted_data = data_with_signals.dropna(subset=['Predicted'])
    if not predicted_data.empty:
        fig.add_trace(
            go.Scatter(x=predicted_data.index,
y=predicted_data['Predicted'],
                      name='Predicted Price', line=dict(color='red',
width=2, dash='dash')),
            row=1, col=1
        )

    if 'BB_Upper' in data_with_signals.columns:
        fig.add_trace(
            go.Scatter(x=data_with_signals.index,
y=data_with_signals['BB_Upper'],
                      name='Upper Band', line=dict(color='gray',
width=1), opacity=0.7),
            row=1, col=1
        )

        fig.add_trace(
            go.Scatter(x=data_with_signals.index,
y=data_with_signals['BB_Lower'],
                      name='Lower Band', line=dict(color='gray',
width=1), opacity=0.7,
                      fill='tonexty',
fillcolor='rgba(128,128,128,0.1)'),
            row=1, col=1
        )

    buy_signals = data_with_signals[data_with_signals['Buy_Signal']]
    if not buy_signals.empty:
        fig.add_trace(
            go.Scatter(x=buy_signals.index, y=buy_signals['Close'],
                      mode='markers', name='Buy Signal',
                      marker=dict(color='green', size=10,
symbol='triangle-up')),
            row=1, col=1
        )

    sell_signals = data_with_signals[data_with_signals['Sell_Signal']]
    if not sell_signals.empty:
        fig.add_trace(
            go.Scatter(x=sell_signals.index, y=sell_signals['Close'],
                      mode='markers', name='Sell Signal',
                      marker=dict(color='red', size=10,
symbol='triangle-down')),
            row=1, col=1
        )
    fig.add_trace(
        go.Bar(x=data_with_signals.index,
y=data_with_signals['Volume'],
               name='Volume', marker_color='rgba(0,100,80,0.6)'),
        row=2, col=1
    )

    fig.update_layout(
        title=f'{company_name} ({ticker_symbol}) - Enhanced Stock
Analysis Dashboard',
        xaxis_title='Date',
        yaxis_title='Price ($)',
        height=800,
        showlegend=True,
        hovermode='x unified'
    )

    fig.update_xaxes(rangeslider_visible=False)
    fig.show()
```

Main Execution

```
def main():
    print("=== Enhanced Stock Price Prediction System ===")

    stock_ticker = get_valid_ticker()
    model_choice = get_model_choice()

    result = fetch_stock_data(stock_ticker)
    if result is None:
        return

    data, company_name = result
    data_with_indicators = add_technical_indicators(data)

    xgb_results = None
    lstm_results = None

    if model_choice in [1, 3]:
        xgb_data = prepare_features_xgboost(data_with_indicators)
        xgb_results = train_xgboost_model(xgb_data)

    if model_choice in [2, 3] and LSTM_AVAILABLE:
        lstm_results = train_lstm_model(data_with_indicators)

    if model_choice == 1 and xgb_results[0] is not None:
        models, X_test, y_test, predictions, split_idx, scaler =
xgb_results
        data_with_signals =
generate_trading_signals(data_with_indicators, predictions, split_idx)
        model_name = "Enhanced Ensemble"

    elif model_choice == 2 and lstm_results[0] is not None:
        model, X_test, y_test, predictions, split_idx, scaler =
lstm_results
        data_with_signals =
generate_trading_signals(data_with_indicators, predictions, split_idx)
        model_name = "Enhanced LSTM"

    elif model_choice == 3:
        if xgb_results[0] is not None and lstm_results[0] is not None:
            xgb_pred = xgb_results[3]
            lstm_pred = lstm_results[3]

            min_len = min(len(xgb_pred), len(lstm_pred))
            xgb_weight = 0.6
            lstm_weight = 0.4
            avg_predictions = (xgb_pred[:min_len] * xgb_weight +
lstm_pred[:min_len] * lstm_weight)

            split_idx = max(xgb_results[4], lstm_results[4])
            data_with_signals =
generate_trading_signals(data_with_indicators, avg_predictions,
split_idx)
            model_name = "Combined Enhanced Models"
        else:
            print("Both models not available. Using available model.")
            if xgb_results[0] is not None:
                models, X_test, y_test, predictions, split_idx, scaler
= xgb_results
                data_with_signals =
generate_trading_signals(data_with_indicators, predictions, split_idx)
                model_name = "Enhanced Ensemble"
            else:
                return

    current_price = data_with_signals['Close'].iloc[-1]
    predicted_price = data_with_signals['Predicted'].iloc[-1] if
pd.notna(data_with_signals['Predicted'].iloc[-1]) else current_price
    current_rsi = data_with_indicators['RSI'].iloc[-1] if 'RSI' in
data_with_indicators.columns else 50

    if 'BB_Upper' in data_with_signals.columns and 'BB_Lower' in
data_with_signals.columns:
        bb_upper = data_with_signals['BB_Upper'].iloc[-1]
        bb_lower = data_with_signals['BB_Lower'].iloc[-1]
        bb_position = (current_price - bb_lower) / (bb_upper -
bb_lower) if bb_upper != bb_lower else 0.5
    else:
        bb_position = 0.5

    buy_prob, hold_prob, sell_prob = calculate_recommendation(
        data_with_signals, current_price, predicted_price,
current_rsi, bb_position
    )

    print(f"\n=== {model_name} Model Results ===")
    print(f"Current Price: ${current_price:.2f}")
    print(f"Predicted Next Price: ${predicted_price:.2f}")
    print(f"Price Change: {((predicted_price - current_price) /
current_price) * 100:.2f}%")
    print(f"Current RSI: {current_rsi:.2f}")

    print(f"\n=== Trading Recommendation ===")
    print(f"BUY Probability: {buy_prob:.1f}%")
    print(f"HOLD Probability: {hold_prob:.1f}%")
    print(f"SELL Probability: {sell_prob:.1f}%")

    if buy_prob > max(hold_prob, sell_prob):
        main_rec = "BUY"
    elif sell_prob > max(buy_prob, hold_prob):
        main_rec = "SELL"
    else:
        main_rec = "HOLD"

    print(f"\nMain Recommendation: {main_rec}")

    create_interactive_plot(data_with_signals, company_name,
stock_ticker)

    print(f"\nEnhanced Analysis complete! Interactive chart displayed
above.")

if __name__ == "__main__":
    main()
```

```
=== Enhanced Stock Price Prediction System ===
Enter a valid stock ticker (e.g., AAPL, TSLA, MSFT): AApl

Choose prediction model:
1. XGBoost (Less time, High accuracy)
2. LSTM (More time, Most accuracy)
3. Both (Conclusion with both models)
Enter your choice (1/2/3): 3

Fetched data for Apple Inc. (AAPL)
Data range: 1980-12-12 to 2025-06-25
Total trading days: 11224

Training Enhanced XGBoost model...
```

```
Ensemble Training RMSE: 0.0627
Ensemble Testing RMSE: 103.8779
Ensemble Training R²: 1.0000
Ensemble Testing R²: -2.2334

Training Enhanced LSTM model...
```

.ipynb to .pkl file convertor

```
!pip uninstall scipy -y

!pip install scipy==1.11.4

import yfinance as yf
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')

from sklearn.preprocessing import MinMaxScaler, RobustScaler
from sklearn.metrics import mean_squared_error, r2_score
import xgboost as xgb
from sklearn.ensemble import RandomForestRegressor,
GradientBoostingRegressor
import pickle

try:
    import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import LSTM, Dense, Dropout,
BatchNormalization, GRU
    from tensorflow.keras.optimizers import Adam
    from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau
    LSTM_AVAILABLE = True
except ImportError:
    LSTM_AVAILABLE = False

def get_valid_ticker():
    while True:
        stock = input("Enter a valid stock ticker (e.g., AAPL, TSLA,
MSFT): ").upper()
        try:
            test = yf.Ticker(stock)
            if test.history(period="1d").empty:
                print("Invalid ticker. Please try again.")
            else:
                return stock
        except Exception as e:
            print(f"Error validating ticker: {e}")
            print("Invalid input. Please try again.")

def get_model_choice():
    print("\nChoose model to train and save:")
    print("1. XGBoost Ensemble")
    print("2. LSTM")
    while True:
        choice = input("Enter your choice (1/2): ")
        if choice in ['1', '2']:
            return int(choice)
        print("Invalid choice. Please enter 1 or 2.")

def fetch_stock_data(ticker_symbol, period="max"):
    ticker = yf.Ticker(ticker_symbol)
    data = ticker.history(period=period)
    if data.empty:
        return None
    info = ticker.info
    company_name = info.get('longName', ticker_symbol)
    print(f"\nFetched data for {company_name} ({ticker_symbol})")
    return data, company_name

def calculate_rsi(prices, window=14):
    delta = prices.diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=window).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=window).mean()
    rs = gain / loss
    return 100 - (100 / (1 + rs))

def calculate_macd(prices, fast=12, slow=26, signal=9):
    ema_fast = prices.ewm(span=fast).mean()
    ema_slow = prices.ewm(span=slow).mean()
    macd = ema_fast - ema_slow
    macd_signal = macd.ewm(span=signal).mean()
    macd_hist = macd - macd_signal
    return macd, macd_signal, macd_hist

def calculate_bollinger_bands(prices, window=20, num_std=2):
    rolling_mean = prices.rolling(window=window).mean()
    rolling_std = prices.rolling(window=window).std()
    bb_upper = rolling_mean + (rolling_std * num_std)
    bb_lower = rolling_mean - (rolling_std * num_std)
    return bb_upper, rolling_mean, bb_lower

def add_technical_indicators(data):
    df = data.copy()
    df['Returns'] = df['Close'].pct_change()
    df['Log_Returns'] = np.log(df['Close'] / df['Close'].shift(1))
    for period in [3, 5, 10, 20, 50, 100]:
        df[f'MA_{period}'] = df['Close'].rolling(window=period).mean()
        df[f'EMA_{period}'] = df['Close'].ewm(span=period).mean()
    df['RSI'] = calculate_rsi(df['Close'])
    df['MACD'], _, _ = calculate_macd(df['Close'])
    df['BB_Upper'], df['BB_Middle'], df['BB_Lower'] =
calculate_bollinger_bands(df['Close'])
    df['BB_Position'] = (df['Close'] - df['BB_Lower']) /
(df['BB_Upper'] - df['BB_Lower'])
    return df

def create_sequence_features(data, sequence_length=10):
    df = data.copy()
    price_cols = ['Close', 'High', 'Low', 'Open', 'Volume']
    indicator_cols = ['RSI', 'MACD', 'BB_Position']
    for col in price_cols + indicator_cols:
        if col in df.columns:
            for i in range(1, sequence_length + 1):
                df[f'{col}_lag_{i}'] = df[col].shift(i)
    return df

def prepare_features_xgboost(data, look_back=20, forecast_horizon=1):
    df = data.copy()
    df = create_sequence_features(df, sequence_length=look_back)
    for i in range(1, forecast_horizon + 1):
        df[f'Target_{i}'] = df['Close'].shift(-i)
    df = df.dropna()
    return df

def train_xgboost_model(data, forecast_horizon=1):
    print("\nTraining Enhanced XGBoost model...")
    target_cols = [f'Target_{i}' for i in range(1, forecast_horizon +
1)]
    feature_cols = [col for col in data.columns if col not in
target_cols + ['Open', 'Close', 'High', 'Low', 'Volume', 'Adj Close']]
    X =
data[feature_cols].select_dtypes(include=[np.number]).fillna(data.mean
())
    y = data[target_cols[0]]
    split_idx = int(len(X) * 0.85)
    X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
    y_train, y_test = y.iloc[:split_idx], y.iloc[split_idx:]
    scaler = RobustScaler()
    X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train),
columns=X_train.columns, index=X_train.index)
    X_test_scaled = pd.DataFrame(scaler.transform(X_test),
columns=X_test.columns, index=X_test.index)
    xgb_model = xgb.XGBRegressor(n_estimators=300, max_depth=6,
learning_rate=0.03, subsample=0.9, colsample_bytree=0.9,
reg_alpha=0.05, reg_lambda=0.05, random_state=42, n_jobs=-1)
```

```
    rf_model = RandomForestRegressor(n_estimators=200, max_depth=10,
min_samples_split=5, min_samples_leaf=2, random_state=42, n_jobs=-1)
    gb_model = GradientBoostingRegressor(n_estimators=200,
max_depth=6, learning_rate=0.05, subsample=0.9, random_state=42)
    models = [('XGB', xgb_model), ('RF', rf_model), ('GB', gb_model)]
    for name, model in models:
        model.fit(X_train_scaled, y_train)
    return models, scaler, X_train_scaled.columns.tolist()

def prepare_lstm_data(data, look_back=60, forecast_horizon=1):
    features = ['Close', 'Volume', 'High', 'Low', 'Open', 'RSI',
'MACD', 'BB_Position']
    feature_data =
data[features].fillna(method='ffill').fillna(method='bfill')
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(feature_data)
    X, y = [], []
    for i in range(look_back, len(scaled_data) - forecast_horizon +
1):
        X.append(scaled_data[i-look_back:i])
        y.append(scaled_data[i:i+forecast_horizon, 0])
    X, y = np.array(X), np.array(y).reshape(-1)
    return X, y, scaler, features

def train_lstm_model(data, forecast_horizon=1):
    if not LSTM_AVAILABLE:
        return None, None, None
    print("\nTraining Enhanced LSTM model...")
    X, y, scaler, features_list = prepare_lstm_data(data,
look_back=60, forecast_horizon=forecast_horizon)
    split_idx = int(len(X) * 0.85)
    X_train, X_test = X[:split_idx], X[split_idx:]
    y_train, y_test = y[:split_idx], y[split_idx:]
    model = Sequential([LSTM(128, return_sequences=True,
input_shape=(X_train.shape[1], X_train.shape[2])), Dropout(0.2),
BatchNormalization(), LSTM(64, return_sequences=True), Dropout(0.2),
BatchNormalization(), GRU(32), Dropout(0.2), Dense(64,
activation='relu'), BatchNormalization(), Dropout(0.1), Dense(32,
activation='relu'), Dense(1)])
    model.compile(optimizer=Adam(learning_rate=0.001), loss='huber',
metrics=['mae'])
    early_stopping = EarlyStopping(monitor='val_loss', patience=20,
restore_best_weights=True)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.7,
patience=10, min_lr=1e-5)
    model.fit(X_train, y_train, batch_size=32, epochs=150,
validation_data=(X_test, y_test), callbacks=[early_stopping,
reduce_lr], verbose=0)
    return model, scaler, features_list
```

```python
def main():
    print("=== Model Training and Saving Script ===")
    stock_ticker = get_valid_ticker()
    model_choice = get_model_choice()
    result = fetch_stock_data(stock_ticker)
    if result is None: return
    data, _ = result
    data_with_indicators = add_technical_indicators(data)
    if model_choice == 1:
        xgb_data = prepare_features_xgboost(data_with_indicators)
        trained_models, scaler, feature_list =
train_xgboost_model(xgb_data)
        artifacts = {'models': trained_models, 'scaler': scaler,
'feature_list': feature_list}
        filename = f'xgb_ensemble_{stock_ticker}.pkl'
        with open(filename, 'wb') as f:
            pickle.dump(artifacts, f)
        print(f"\n XGBoost Ensemble model and artifacts saved to
'{filename}'")
    elif model_choice == 2 and LSTM_AVAILABLE:
        trained_model, scaler, feature_list =
train_lstm_model(data_with_indicators)
        if trained_model:
            model_filename = f'lstm_model_{stock_ticker}.keras'
            trained_model.save(model_filename)
            print(f"\n LSTM model saved to '{model_filename}'")
            artifacts = {'scaler': scaler, 'feature_list':
feature_list}
            scaler_filename = f'lstm_artifacts_{stock_ticker}.pkl'
            with open(scaler_filename, 'wb') as f:
                pickle.dump(artifacts, f)
            print(f"LSTM scaler and feature list saved to
'{scaler_filename}'")

if __name__ == "__main__":
    main()
```