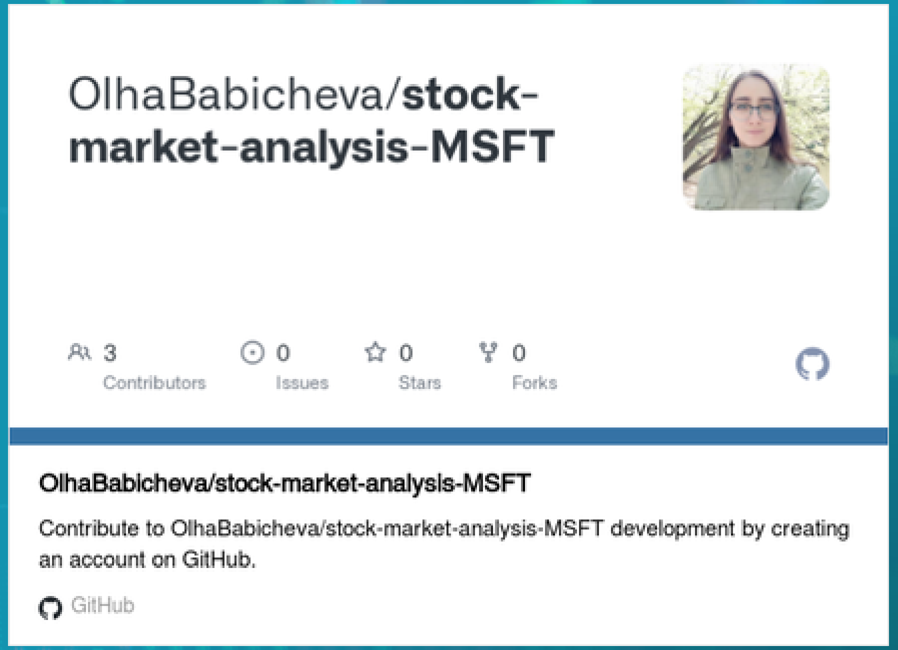
STOCK MARKET ANALYSIS

Of Microsoft



Olha Babicheva
Anna Saldat
Agata Jabłońska

Roadmap

Planning

Selection of project topic and core tools.
Assignment of responsibilities among team members.

Data preparation

Configuration and downloading of historical MSFT stock data from Yahoo Finance.
Setting up the repository and GitHub integration.

Feature engineering

Calculating technical indicators (MA, RSI, MACD, etc.) from historical data.
Saving processed datasets in .csv format for training and testing.

Training

Loading training and test datasets and verifying format correctness.
Training multiple regression models (Linear Regression, Random Forest, SVR, Neural Network).

Model evaluation

Evaluating trained models using regression metrics.
Saving trained models and scaler into a bundle.

CI pipeline (GitHub Actions)

Automated execution of data preparation and model training on push, pull request, or manual triggers.
Verification of repository history, branch changes, and model outputs.

Front End Project

Configuration and deployment of a user-friendly web interface.
Displaying live stock predictions in a clear and readable format.

Database **YahooFinance**

Necessary **libraries**

- pandas
- scikit-learn
- yfinance
- streamlit (used in next phases)

Implemented **functions**

```python
import yfinance as yf # Yahoo Finance API downloader (fetches historical market data)
import numpy as np # Numerical library (used for mathematical operations)

# --- CONFIGURATION ---
TICKER = 'MSFT'
START_DATE = '2019-01-01'
TEST_SIZE_RATIO = 0.2
TRAIN_FILE = 'train.csv'
TEST_FILE = 'test.csv'
```

```python
def download_and_prepare_data(ticker, start_date, test_ratio):
    """
    Downloads historical stock data, performs feature engineering, and splits it
    chronologically into training and testing sets.
    """
    print(f"1. Downloading historical data for {ticker} (Start Date: {start_date})...")

    try:
        data = yf.download(ticker, start=start_date, end=None, progress=False)
    except Exception as e:
        print(f"Error downloading data: {e}")
        return None

    if data.empty:
        print("No data retrieved. Exiting data pipeline.")
        return None

    print(f"Download successful. Total samples: {len(data)}")
    print("2. Feature Engineering: Calculating Moving Averages and setting target...")

    # Get the Close prices
    close_prices = data['Close'].copy()

    # Feature 1 & 2: Simple Moving Averages
    data['MA_10'] = close_prices.rolling(window=10).mean()
    data['MA_30'] = close_prices.rolling(window=30).mean()

    # Feature 3: Daily Range (Measure of volatility)
    data['Daily_Range'] = data['High'] - data['Low']
```

data_preparation.py

```python
    # Feature 4 & 5: Daily Returns
    #(returns capture day to day relative price changes)
    data['Return_1d'] = close_prices.pct_change()
    # Log returns are time-additive and commonly assumed to be closer to normality
    data['Log_Return_1d'] = np.log(close_prices / close_prices.shift(1))

    # Feature 6 & 7: Momentum
    # (measures medium-term trend persistence.
    # Positive values indicate upward pressure, negative values downward pressure)
    data['Momentum_5'] = close_prices - close_prices.shift(5)
    data['Momentum_10'] = close_prices - close_prices.shift(10)

    # Feature 8: RSI (Relative Strength Index)
    # RSI measures the speed and magnitude of recent price movements
    # Values >70 often indicate overbought conditions, Values <30 often indicate oversold conditions

    delta = close_prices.diff()
    # Separate positive and negative price changes
    gain = delta.clip(lower=0)
    loss = -delta.clip(upper=0)
    # Rolling averages of gains and losses (standard 14-day window)
    avg_gain = gain.rolling(14).mean()
    avg_loss = loss.rolling(14).mean()
    # Relative Strength and RSI formula
    rs = avg_gain / avg_loss
    data['RSI_14'] = 100 - (100 / (1 + rs))

    # Feature 9 & 10: MACD Indicator - trend-following momentum indicator
    # (MACD captures the relationship between short-term and long-term trends)

    # Short-term exponential moving average
    ema_12 = close_prices.ewm(span=12, adjust=False).mean()
    # Long-term exponential moving average
    ema_26 = close_prices.ewm(span=26, adjust=False).mean()
```

Implemented **functions**

data_preparation.py

Implemented **functions**

```python
    # MACD line
    data['MACD'] = ema_12 - ema_26
    # Signal line (EMA of MACD)
    data['MACD_Signal'] = data['MACD'].ewm(span=9, adjust=False).mean()

    # Target Variable: Predict the next day's Close price (Next_Close)
    data['Next_Close'] = close_prices.shift(-1)

    # Define the features we will use for the model
    FEATURES = [
        'Close', 'MA_10', 'MA_30',
        'Volume', 'Daily_Range', 'Return_1d',
        'Log_Return_1d', 'Momentum_5', 'Momentum_10',
        'RSI_14', 'MACD', 'MACD_Signal'
    ]

    # Drop rows that have NaN values due to the initial 30-day rolling
    # window and the last target row
    df = data.dropna()

    print(f"Data cleaned. Usable data points: {len(df)}")

    # Calculate the index for the chronological split
    train_size = int(len(df) * (1 - test_ratio))

    # Split the dataframes
    df_train = df.iloc[:train_size].copy()
    df_test = df.iloc[train_size:].copy()

    return df_train, df_test, FEATURES
```

data_preparation.py

Implemented **functions**

```python
def save_data(df_train, df_test, features):
    """Saves the train and test DataFrames, including the target and features, to CSV."""

    # Columns to save: Target & Features
    cols_to_save = ['Next_Close'] + features

    # Save to CSV files
    df_train[cols_to_save].to_csv(TRAIN_FILE)
    df_test[cols_to_save].to_csv(TEST_FILE)

    print("3. Data split and saved to CSV files:")
    print(f"Training set size: {len(df_train)} samples ({TRAIN_FILE})")
    print(f"Testing set size: {len(df_test)} samples ({TEST_FILE})")

    return features


if __name__ == '__main__':
    data_output = download_and_prepare_data(TICKER, START_DATE, TEST_SIZE_RATIO)

    if data_output:
        df_train, df_test, features_list = data_output
        save_data(df_train, df_test, features_list)
```

data_preparation.py

Implemented **functions**

```python
import os # Standard library for interacting with the operating system (checking file paths)
import pandas as pd # Data manipulation library
import numpy as np # Numerical library (used for mathematical operations)
import joblib  # Library for saving and loading models
from sklearn.linear_model import LinearRegression, Ridge  # Ridge adds L2 regularization to Linear Regression
from sklearn.ensemble import RandomForestRegressor # A bagging-based ensemble model (often more stable than boosting)
from sklearn.svm import SVR # Support Vector Regression (good for non-linear patterns)
from sklearn.neural_network import MLPRegressor # Multi-layer Perceptron (Neural Network) regressor
from sklearn.preprocessing import StandardScaler # Crucial for models like SVR and Ridge
from sklearn.metrics import mean_squared_error, r2_score # Functions to calculate model accuracy

# --- CONFIGURATION ---
TRAIN_FILE = 'train.csv'
TEST_FILE = 'test.csv'
MODEL_EXPORT_FILE = 'models_bundle.joblib'
FEATURES = [
    'Close', 'MA_10', 'MA_30',
    'Volume', 'Daily_Range', 'Return_1d',
    'Log_Return_1d', 'Momentum_5', 'Momentum_10',
    'RSI_14', 'MACD', 'MACD_Signal'
]
TARGET = 'Next_Close'
```

training.py

Implemented **functions**

```python
def load_data(train_path, test_path):
    """Loads the training and testing datasets from CSV files."""
    if not os.path.exists(train_path) or not os.path.exists(test_path):
        print(f"Error: One or both data files ({train_path}, {test_path}) not found.")
        print("Please run 'data_preparation.py' first.")
        return None, None

    print("1. Loading data from CSV files.")
    COL_NAMES = ['Date', TARGET] + FEATURES
    df_train = pd.read_csv(train_path,
                           index_col=0,
                           parse_dates=True,
                           skiprows=3,
                           header=None,
                           names=COL_NAMES)

    df_test = pd.read_csv(test_path,
                          index_col=0,
                          parse_dates=True,
                          skiprows=3,
                          header=None,
                          names=COL_NAMES)

    print(f"Train samples loaded: {len(df_train)}")
    print(f"Test samples loaded: {len(df_test)}")

    # Ensure all required features and the target exist
    if not all(col in df_train.columns for col in FEATURES + [TARGET]):
        print("Error: Missing required columns in loaded data. Check FEATURES and TARGET lists.")
        return None, None

    return df_train, df_test
```

training.py

Implemented **functions**

```python
def train_and_evaluate(df_train, df_test, features, target):
    """Trains multiple models to predict the Next_Close price."""

    # Prepare data for sklearn
    X_train_raw = df_train[features].values
    y_train = df_train[target].values
    X_test_raw = df_test[features].values
    y_test = df_test[target].values

    # Scaling is mandatory for SVR and Ridge to perform correctly
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train_raw)
    X_test = scaler.transform(X_test_raw)

    models = {
        "Linear Regression": LinearRegression(),
        "Ridge (L2)": Ridge(alpha=10.0), # Penalizes large coefficients to reduce noise sensitivity
        "Random Forest": RandomForestRegressor(
            n_estimators=400, # Number of trees
            random_state=42
        ), # Stable bagging
        "Neural Network (MLP)": MLPRegressor(
            hidden_layer_sizes=(10, 64, 64), # Three hidden layers with 10, 64 and 64 neurons
            activation='relu',  # Rectified Linear Unit activation function
            solver='adam',      # Optimizer for weight optimization
            max_iter=1000,      # Maximum number of iterations
            random_state=42     # For reproducible results
        ),
        "SVR (RBF Kernel)": SVR(
            kernel='linear',
            epsilon=0.1 # Threshold where no penalty is given to errors
        ), # Non-linear boundary mapping
    }

    results = []
    trained_models = {}
```

training.py

Implemented **functions**

```python
    print(f"2. Training models to predict absolute {target}...")

    for name, model in models.items():
        # Training directly on the absolute price
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        # Evaluation
        mse = mean_squared_error(y_test, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_test, y_pred)
        results.append({
            "Model": name,
            "MSE": round(mse, 4),
            "RMSE (USD)": round(rmse, 4),
            "R2 Score": round(r2, 4)
        })
        trained_models[name] = model

    # Save the bundle: models, scaler, and feature list
    bundle = {
        'models': trained_models,
        'scaler': scaler,
        'features': features
    }
    joblib.dump(bundle, MODEL_EXPORT_FILE)
    print(f"3. Models and scaler saved to {MODEL_EXPORT_FILE}")

    print("="*50)
    print(pd.DataFrame(results).sort_values(by="RMSE (USD)").to_string(index=False))
    print("="*50)

    comparison_df = pd.DataFrame(results)
    print("="*50)
    print(f"MODEL COMPARISON SUMMARY (Target: {target})")
    print("="*50)
    print(comparison_df.sort_values(by="RMSE (USD)").to_string(index=False))
    print("="*50)

    return comparison_df
```

training.py

Implemented **functions**

```python
if __name__ == '__main__':
    # Execute the loading and training workflow
    df_train_data, df_test_data = load_data(TRAIN_FILE, TEST_FILE)

    if df_train_data is not None:
        train_and_evaluate(df_train_data, df_test_data, FEATURES, TARGET)

```

training.py

Implemented **functions**

```python
from datetime import datetime, timedelta
import streamlit as st
import numpy as np
import yfinance as yf
import joblib

# --- APP CONFIGURATION ---
TICKER = 'MSFT'
MODEL_PATH = 'models_bundle.joblib'

st.set_page_config(page_title="MSFT Price Predictor", layout="wide")

@st.cache_resource
def load_bundle():
    """Loads the saved models and scaler."""
    try:
        return joblib.load(MODEL_PATH)
    except:
        return None
```

```python
def get_latest_data(ticker):
    """Fetches and prepares the most recent data for prediction."""
    # Fetch enough data to calculate technical indicators (approx 60 days)
    end_date = datetime.now()
    start_date = end_date - timedelta(days=60)
    try:
        data = yf.download(ticker, start=start_date, end=None, progress=False)
    except Exception as e:
        print(f"Error downloading data: {e}")
        return None

    if data.empty:
        return None

    # Feature Engineering (Mirrors data_preparation.py)
    # Get the Close prices
    close_prices = data['Close'].copy()

    # Feature 1 & 2: Simple Moving Averages
    data['MA_10'] = close_prices.rolling(window=10).mean()
    data['MA_30'] = close_prices.rolling(window=30).mean()

    # Feature 3: Daily Range (Measure of volatility)
    data['Daily_Range'] = data['High'] - data['Low']

    # Feature 4 & 5: Daily Returns
    data['Return_1d'] = close_prices.pct_change()
    data['Log_Return_1d'] = np.log(close_prices / close_prices.shift(1))

    # Feature 6 & 7: Momentum
    data['Momentum_5'] = close_prices - close_prices.shift(5)
    data['Momentum_10'] = close_prices - close_prices.shift(10)

```

app.py

Implemented **functions**

```python
    # Feature 8: RSI (Relative Strength Index)
    delta = close_prices.diff()
    gain = delta.clip(lower=0)
    loss = -delta.clip(upper=0)
    avg_gain = gain.rolling(14).mean()
    avg_loss = loss.rolling(14).mean()
    rs = avg_gain / avg_loss
    data['RSI_14'] = 100 - (100 / (1 + rs))

    # Feature 9 & 10: MACD Indicator
    ema_12 = close_prices.ewm(span=12, adjust=False).mean()
    ema_26 = close_prices.ewm(span=26, adjust=False).mean()
    data['MACD'] = ema_12 - ema_26
    data['MACD_Signal'] = data['MACD'].ewm(span=9, adjust=False).mean()

    return data.dropna()

# --- UI LAYOUT ---
st.title("📈 Microsoft (MSFT) Stock Price Predictor")
st.markdown("This app provides statistical predictions for the **Next Day Closing Price** using pre-trained ML models.")

bundle = load_bundle()

# Check if the bundle exists; if not, show an error message
if bundle is None:
    st.error("Model bundle not found. Please run `training.py` first to generate `models_bundle.joblib`.")
else:
    # Fetch the most recent live data for MSFT
    latest_df = get_latest_data(TICKER)

    if latest_df is not None:
        # Extract the very last available row of data to use for the prediction
        current_data = latest_df.iloc[[-1]]
        # Format the date and price for display
        last_date = current_data.index[0].strftime('%Y-%m-%d')
        last_price = float(current_data['Close'].iloc[0])

        # Create three columns for the summary metrics row
        col1, col2, col3 = st.columns(3)
```

app.py

Implemented **functions**

```python
    with col1:
        st.subheader("Market Status")
        # Show the most recent closing price
        st.metric("Last Known Close", f"${last_price:,.2f}")
        st.caption(f"Data as of: {last_date}")

    with col2:
        st.subheader("Model Selection")
        # Allow the user to choose between different trained models (e.g., Random Forest, Linear Regression)
        selected_model_name = st.selectbox("Choose Model", list(bundle['models'].keys()))

    with col3:
        st.subheader("Volatility Index")
        # Show the most recent intraday price fluctuation
        daily_range = float(current_data['Daily_Range'].iloc[0])
        st.metric("Intraday Range", f"${daily_range:.2f}")

    # Prediction Logic
    # 1. Get the exact list of features the model was trained on
    features_to_use = bundle['features']
    # 2. Extract those values from our current data row
    X_raw = current_data[features_to_use].values
    # 3. Apply the saved scaler to normalize the features
    X_scaled = bundle['scaler'].transform(X_raw)

    # 4. Use the selected model to predict the next closing price
    model = bundle['models'][selected_model_name]
    prediction_val = model.predict(X_scaled)
    prediction = float(prediction_val[0])

    # Add a visual horizontal line
    st.divider()

    # Display Prediction Statistics
    st.subheader(f"Statistical Analysis: {selected_model_name}")
    # Calculate the dollar difference and percentage change from the last known price
    diff = prediction - last_price
    percent = (diff / last_price) * 100
```

app.py

Implemented **functions**

```python

        # Create two columns for results
        res_col1, res_col2 = st.columns(2)
        with res_col1:
            # Show the predicted price with a delta (green/red arrow)
            st.metric(
                label="Predicted Next Close",
                value=f"${prediction:,.2f}",
                delta=f"{diff:+.2f} ({percent:+.2f}%)"
            )

        with res_col2:
            # Determine market sentiment based on whether the prediction is up or down
            sentiment = "Bullish" if diff > 0 else "Bearish"
            st.info(f"Signal: **{sentiment}** | Expected change of **{percent:.2f}%**")

        # Create an expandable section to let users inspect the raw math behind the prediction
        with st.expander("View Raw Feature Data (Last 10 Days)"):
            st.dataframe(latest_df.tail(10))

    else:
        # Show error if yfinance fails or returns nothing
        st.error("Failed to fetch recent data from Yahoo Finance.")

# Add a permanent disclaimer in the sidebar
st.sidebar.info("Disclaimer: Statistical models are for informational purposes only. Trading involves significant risk.")

```

app.py