XGBoost Stock Price Prediction

Adapted for CSV input with automatic feature engineering

```
# Install dependencies (run once)
%pip install pandas numpy scikit-learn xgboost joblib -q
```

```
import time
import pandas as pd
import numpy as np
import joblib
from datetime import datetime
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error
from sklearn.preprocessing import MinMaxScaler
from xgboost import XGBRegressor
import warnings
warnings.filterwarnings('ignore')
```

Configuration

```
# ===== EDIT THESE =====
CSV_PATH = '/content/aapl.csv'  # Path to your CSV file
PREDICTION_HORIZON = 60  # Predict N periods ahead
TRAIN_RATIO = 0.8  # 80% train, 20% test
OUTPUT_DIR = './model_artifacts'  # Where to save model

# XGBoost Parameters
XGB_PARAMS = {
    'n_estimators': 300,
    'max_depth': 100,
    'learning_rate': 0.1,
    'objective': 'reg:squarederror',
    'alpha': 10,
    'tree_method': 'hist',  # Use 'hist' for CPU
    # 'device': 'cuda',  # Uncomment for GPU
    'random_state': 42,
}
```

Feature Engineering Functions

```
def create_technical_features(df):
    """
    Create technical indicator features from OHLCV data.
    """
    df = df.copy()

    # --- Basic Price Features ---
    df['return_1'] = df['close'].pct_change(1)
    df['return_5'] = df['close'].pct_change(5)
    df['return_10'] = df['close'].pct_change(10)
    df['return_20'] = df['close'].pct_change(20)

    # --- Moving Averages ---
    for window in [5, 10, 20, 50, 100]:
        df[f'sma_{window}'] = df['close'].rolling(window=window).mean()
        df[f'ema_{window}'] = df['close'].ewm(span=window, adjust=False).mean()

    # --- Price relative to MAs ---
    df['close_to_sma_20'] = df['close'] / df['sma_20']
    df['close_to_sma_50'] = df['close'] / df['sma_50']
    df['sma_20_to_sma_50'] = df['sma_20'] / df['sma_50']

    # --- Volatility Features ---
    df['volatility_5'] = df['return_1'].rolling(window=5).std()
    df['volatility_10'] = df['return_1'].rolling(window=10).std()
    df['volatility_20'] = df['return_1'].rolling(window=20).std()

    # --- High-Low Range ---
    df['hl_range'] = (df['high'] - df['low']) / df['close']
    df['hl_range_ma_10'] = df['hl_range'].rolling(window=10).mean()

    # --- Price Position within Range ---
    df['close_position'] = (df['close'] - df['low']) / (df['high'] - df['low'] + 1e-8)

    # --- Open-Close Relationship ---
    df['oc_range'] = (df['close'] - df['open']) / df['open']
    df['body_to_range'] = (df['close'] - df['open']) / (df['high'] - df['low'] + 1e-8)

    # --- Volume Features ---
    df['volume_ma_10'] = df['volume'].rolling(window=10).mean()
    df['volume_ma_20'] = df['volume'].rolling(window=20).mean()
    df['volume_ratio'] = df['volume'] / (df['volume_ma_20'] + 1)
    df['volume_change'] = df['volume'].pct_change(1)

    # --- RSI ---
    delta = df['close'].diff()
    gain = delta.where(delta > 0, 0).rolling(window=14).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
    rs = gain / (loss + 1e-8)
    df['rsi_14'] = 100 - (100 / (1 + rs))

    # --- Rate of Change ---
    df['roc_5'] = (df['close'] - df['close'].shift(5)) / df['close'].shift(5)
    df['roc_10'] = (df['close'] - df['close'].shift(10)) / df['close'].shift(10)
    df['roc_20'] = (df['close'] - df['close'].shift(20)) / df['close'].shift(20)

    # --- MACD ---
    ema_12 = df['close'].ewm(span=12, adjust=False).mean()
    ema_26 = df['close'].ewm(span=26, adjust=False).mean()
    df['macd'] = ema_12 - ema_26
    df['macd_signal'] = df['macd'].ewm(span=9, adjust=False).mean()
    df['macd_hist'] = df['macd'] - df['macd_signal']

    # --- Bollinger Bands ---
    df['bb_middle'] = df['close'].rolling(window=20).mean()
    bb_std = df['close'].rolling(window=20).std()
    df['bb_upper'] = df['bb_middle'] + 2 * bb_std
    df['bb_lower'] = df['bb_middle'] - 2 * bb_std
    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'] + 1e-8)

    # --- Lag Features ---
    for lag in [1, 2, 3, 5, 10]:
        df[f'close_lag_{lag}'] = df['close'].shift(lag)
        df[f'return_lag_{lag}'] = df['return_1'].shift(lag)

    # --- Time-based Features ---
    df['hour'] = df['date'].dt.hour
    df['dayofweek'] = df['date'].dt.dayofweek
    df['is_market_open'] = ((df['hour'] >= 9) & (df['hour'] < 16)).astype(int)

    return df
```

Load and Prepare Data

```
# Load CSV
df = pd.read_csv(CSV_PATH)
print(f"Loaded: {len(df)} rows")
print(f"Columns: {df.columns.tolist()}")
df.head()
```

```
Loaded: 215960 rows
Columns: ['date', 'open', 'high', 'low', 'close', 'volume']
```

	date	open	high	low	close	volume
0	2023-07-03 04:00:00	193.99	194.25	193.97	194.13	24142
1	2023-07-03 04:00:00	193.99	194.25	193.97	194.13	24142
2	2023-07-03 04:05:00	194.06	194.16	193.99	194.00	8566

```
# Parse datetime and clean
df['date'] = pd.to_datetime(df['date'])
df = df.drop_duplicates(subset=['date']).reset_index(drop=True)
df = df.sort_values('date').reset_index(drop=True)
print(f"After dedup: {len(df)} rows")
print(f"Date range: {df['date'].min()} to {df['date'].max()}")
```

```
After dedup: 108460 rows
Date range: 2023-07-03 04:00:00 to 2025-10-24 19:55:00
```

```
# Create features
print("Engineering features...")
df = create_technical_features(df)

# Create target (future close price)
df['target'] = df['close'].shift(-PREDICTION_HORIZON)

# Clean up
df = df.replace([np.inf, -np.inf], np.nan)
df = df.dropna().reset_index(drop=True)
print(f"After cleaning: {len(df)} rows")
```

```
Engineering features...
After cleaning: 107253 rows
```

```
# Check features
print(f"Total columns: {len(df.columns)}")
df.head()
```

```
Total columns: 61
```

	date	open	high	low	close	volume	return_1	return_5	return_10	return_20	...	close_lag_3	return_
0	2023-07-03 12:30:00	192.110	192.140	192.0397	192.0400	347019	-0.000390	-0.000598	-0.000858	0.000951	...	192.210	0.0
1	2023-07-03 12:35:00	192.040	192.125	192.0100	192.1000	362782	0.000312	-0.000311	-0.001204	0.000630	...	192.120	-0.0
2	2023-07-03 12:40:00	192.100	192.330	192.1000	192.2255	550794	0.000653	0.000081	-0.000283	0.001122	...	192.115	-0.0
3	2023-07-03 12:45:00	192.225	192.320	192.1199	192.2450	438549	0.000101	0.000651	0.000078	0.001015	...	192.040	-0.0
4	2023-07-03 12:50:00	192.250	192.395	192.2250	192.3650	618701	0.000624	0.001301	0.000390	0.000291	...	192.100	0.0

5 rows × 61 columns

Prepare Train/Test Split

```
# Define feature columns
exclude_cols = ['date', 'target', 'open', 'high', 'low', 'close', 'volume']
feature_cols = [c for c in df.columns if c not in exclude_cols]
print(f"Using {len(feature_cols)} features")
print(feature_cols)
```

```
Using 54 features
['return_1', 'return_5', 'return_10', 'return_20', 'sma_5', 'ema_5', 'sma_10', 'ema_10', 'sma_20', 'ema_20', 'sma_50'
```

```
# Time-based split
split_idx = int(len(df) * TRAIN_RATIO)
train_df = df.iloc[:split_idx]
test_df = df.iloc[split_idx:]

print(f"Train: {len(train_df)} rows ({train_df['date'].min()} to {train_df['date'].max()})")
print(f"Test:  {len(test_df)} rows ({test_df['date'].min()} to {test_df['date'].max()})")
```

```
Train: 85802 rows (2023-07-03 12:30:00 to 2025-05-08 15:15:00)
Test:  21451 rows (2025-05-08 15:20:00 to 2025-10-24 14:55:00)
```

```
# Prepare X and y
X_train = train_df[feature_cols]
y_train = train_df['target']
X_test = test_df[feature_cols]
y_test = test_df['target']

print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
```

```
X_train shape: (85802, 54)
X_test shape: (21451, 54)
```

```
# Scale features
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
print("Features scaled")
```

```
Features scaled
```

Train Model

```
# Initialize model
model = XGBRegressor(**XGB_PARAMS)
print("Model parameters:")
print(XGB_PARAMS)
```

```
Model parameters:
{'n_estimators': 300, 'max_depth': 100, 'learning_rate': 0.1, 'objective': 'reg:squarederror', 'alpha': 10, 'tree_met
```

```
# Train
print("Training...")
start_time = time.time()

model.fit(
    X_train_scaled, y_train,
    eval_set=[(X_test_scaled, y_test)],
    verbose=50
)

train_duration = time.time() - start_time
print(f"\nTraining completed in {train_duration:.2f} seconds")
```

```
Training...
[0]     validation_0-rmse:25.95580
[50]    validation_0-rmse:3.05052
[100]   validation_0-rmse:3.03502
[150]   validation_0-rmse:3.03450
[200]   validation_0-rmse:3.03454
[250]   validation_0-rmse:3.03449
[299]   validation_0-rmse:3.03460

Training completed in 314.65 seconds
```

Evaluate

```
# Predict
y_pred_train = model.predict(X_train_scaled)
y_pred_test = model.predict(X_test_scaled)
```

```
# Metrics
train_rmse = np.sqrt(mean_squared_error(y_train, y_pred_train))
test_rmse = np.sqrt(mean_squared_error(y_test, y_pred_test))
train_mape = mean_absolute_percentage_error(y_train, y_pred_train) * 100
test_mape = mean_absolute_percentage_error(y_test, y_pred_test) * 100

print("=" * 50)
print("EVALUATION METRICS")
print("=" * 50)
print(f"Train RMSE: ${train_rmse:.4f}")
print(f"Test RMSE:  ${test_rmse:.4f}")
print(f"Train MAPE: {train_mape:.2f}%")
print(f"Test MAPE:  {test_mape:.2f}%")
```

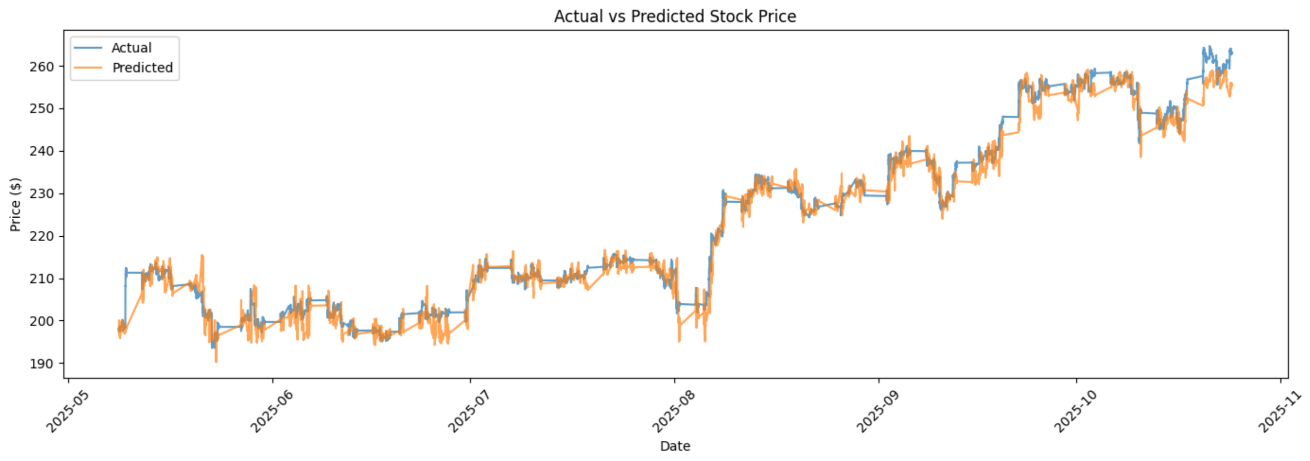
```
==================================================
EVALUATION METRICS
==================================================
Train RMSE: $0.1435
Test RMSE:  $3.0346
Train MAPE: 0.05%
Test MAPE:  0.98%
```

```
# Results dataframe
results = pd.DataFrame({
    'datetime': test_df['date'].values,
    'actual': y_test.values,
    'predicted': y_pred_test,
    'difference': np.abs(y_test.values - y_pred_test),
})
results.head(10)
```

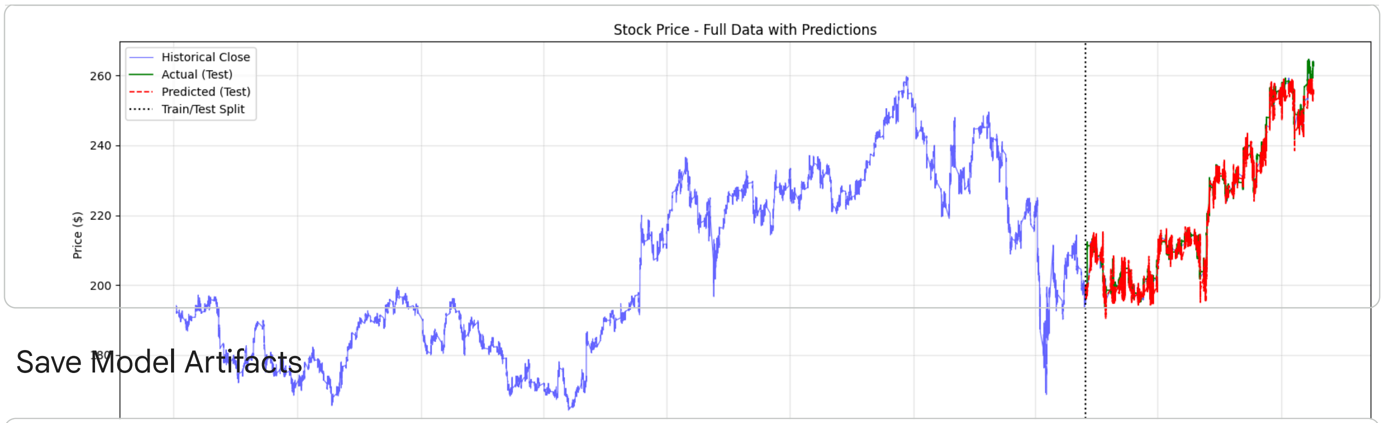
	datetime	actual	predicted	difference
0	2025-05-08 15:20:00	197.99	197.961029	0.028971
1	2025-05-08 15:25:00	197.98	198.236374	0.256374
2	2025-05-08 15:30:00	198.07	198.005371	0.064629
3	2025-05-08 15:35:00	198.00	197.363419	0.636581
4	2025-05-08 15:40:00	198.03	197.109497	0.920503
5	2025-05-08 15:45:00	197.94	197.374786	0.565214
6	2025-05-08 15:50:00	197.92	198.911636	0.991636
7	2025-05-08 15:55:00	198.00	199.217819	1.217819
8	2025-05-08 16:00:00	198.04	199.704971	1.664971
9	2025-05-08 16:05:00	198.13	200.066162	1.936162

Next steps: Generate code with results New interactive sheet

```
# Plot (optional)
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 5))
plt.plot(results['datetime'], results['actual'], label='Actual', alpha=0.7)
plt.plot(results['datetime'], results['predicted'], label='Predicted', alpha=0.7)
plt.title('Actual vs Predicted Stock Price')
plt.xlabel('Date')
plt.ylabel('Price ($)')
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



```
# ===== FULL DATA + PREDICTIONS PLOT =====
import matplotlib.pyplot as plt

# Load original CSV for full history
df_full = pd.read_csv(CSV_PATH)
df_full['date'] = pd.to_datetime(df_full['date'])
df_full = df_full.drop_duplicates(subset=['date']).sort_values('date')

# Plot
fig, ax = plt.subplots(figsize=(16, 6))

# Full historical close price
ax.plot(df_full['date'], df_full['close'], label='Historical Close', color='blue', alpha=0.6, linewidth=0.8)

# Actual vs Predicted on test set
ax.plot(results['datetime'], results['actual'], label='Actual (Test)', color='green', linewidth=1.2)
ax.plot(results['datetime'], results['predicted'], label='Predicted (Test)', color='red', linewidth=1.2, lines

# Mark train/test split
split_date = results['datetime'].min()
ax.axvline(x=split_date, color='black', linestyle=':', linewidth=1.5, label=f'Train/Test Split')

ax.set_title('Stock Price - Full Data with Predictions')
ax.set_xlabel('Date')
ax.set_ylabel('Price ($)')
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Save Model Artifacts

```
import os
os.makedirs(OUTPUT_DIR, exist_ok=True)

# Save model
model.save_model(f"{OUTPUT_DIR}/xgboost_model.json")
print(f"Model saved: {OUTPUT_DIR}/xgboost_model.json")

# Save scaler
joblib.dump(scaler, f"{OUTPUT_DIR}/scaler.joblib")
print(f"Scaler saved: {OUTPUT_DIR}/scaler.joblib")

# Save feature names
joblib.dump(feature_cols, f"{OUTPUT_DIR}/feature_names.joblib")
print(f"Feature names saved: {OUTPUT_DIR}/feature_names.joblib")

# Save config
config = {'prediction_horizon': PREDICTION_HORIZON, 'train_ratio': TRAIN_RATIO}
joblib.dump(config, f"{OUTPUT_DIR}/config.joblib")
print(f"Config saved: {OUTPUT_DIR}/config.joblib")

# Save predictions
results.to_csv(f"{OUTPUT_DIR}/predictions.csv", index=False)
print(f"Predictions saved: {OUTPUT_DIR}/predictions.csv")
```

```
Model saved: ./model_artifacts/xgboost_model.json
Scaler saved: ./model_artifacts/scaler.joblib
Feature names saved: ./model_artifacts/feature_names.joblib
Config saved: ./model_artifacts/config.joblib
Predictions saved: ./model_artifacts/predictions.csv
```

Test Loading Model (for inference)

```
# Load and verify
loaded_model = XGBRegressor()
loaded_model.load_model(f"{OUTPUT_DIR}/xgboost_model.json")
loaded_scaler = joblib.load(f"{OUTPUT_DIR}/scaler.joblib")
loaded_features = joblib.load(f"{OUTPUT_DIR}/feature_names.joblib")

# Quick test
test_pred = loaded_model.predict(loaded_scaler.transform(X_test.iloc[:5]))
print("Model loaded successfully!")
print(f"Sample predictions: {test_pred}")
```

```
Model loaded successfully!
Sample predictions: [197.96103 198.23637 198.00537 197.36342 197.1095 ]
```

Done!

Artifacts saved in `model_artifacts/`:

- `xgboost_model.json` - trained model
- `scaler.joblib` - feature scaler
- `feature_names.joblib` - feature column names

- `config.joblib` - configuration

Use these in your FastAPI inference service.