StockDataProcessor: Advanced Stock Data Processing Library

GITHUB :

https://github.com/quantcommunitynitrkl/data_preprocessing_OHLCV.git

Introduction

StockDataProcessor is a comprehensive Python library designed for financial data analysts, quantitative traders, and data scientists working with stock market time-series data. Built on top of robust libraries like `pandas`, `yfinance`, `scikit-learn`, and `matplotlib`, it provides a unified interface for downloading historical stock data, handling missing values and outliers, performing advanced imputations, and generating insightful visualizations.

This library addresses common pain points in stock data workflows:

- **Data Acquisition**: Seamless download from Yahoo Finance.
- **Data Cleaning**: Intelligent filling of missing dates and NaN values using statistical, machine learning, and smoothing techniques.
- **Outlier Management**: Multi-method detection and treatment to ensure data integrity.
- **Visualization**: Flexible plotting for exploratory analysis, including candlestick charts and interactive Plotly figures.

Whether you're backtesting trading strategies, building predictive models, or conducting market research, **StockDataProcessor** streamlines your pipeline into a modular, extensible class.

Key Features

- **Modular Design**: Static methods for standalone use or instance-based workflows.
- **Advanced Imputation**: Supports KNN, Markov chains, Kalman filters, and more.
- **Outlier Handling**: Seven detection methods with customizable combination strategies.
- **Rich Visualizations**: 15+ graph types, from basic histograms to choropleth maps.
- **Test-Friendly**: Built-in sample dataset for quick prototyping.
- **Error-Resilient**: Extensive input validation and graceful fallbacks.

Installation

Prerequisites

- Python 3.8 or higher.
- Access to the internet for data downloads (via `yfinance`).

Dependencies

Install via pip:

```
pip install yfinance pandas numpy scikit-learn matplotlib seaborn mplfinance plotly scipy pykalman
```

For development (optional):

```
pip install -r requirements-dev.txt  # Includes pytest, black, etc.
```

Note: `pykalman` requires NumPy; ensure it's installed for Kalman filter features.

Quick Start

```python
import pandas as pd
from stockdataprocessor import StockDataProcessor  # Assuming the class is in a module named stockdataprocessor

# Download data
df = StockDataProcessor.download_stock_data("AAPL", period="6mo")

# Fill missing dates
df = StockDataProcessor.fill_missing_dates(df)

# Advanced NaN filling (example config)
col_tech_map = {"close": [("ffill", {}), ("sma", {"window": 5})]}
df = StockDataProcessor.fill_nan_advanced(df, col_tech_map)

# Plot a candlestick chart
StockDataProcessor.plot_graph(df, columns=["open", "high", "low", "close"], graph_type="candlestick")

# Detect and treat outliers
outliers = StockDataProcessor.detect_outliers_advanced(df, numeric_cols=["close", "volume"])
treatment_map = {"close": [("median_replace", {})]}
df_treated = StockDataProcessor.treat_outliers(df, outliers, treatment_map)
```

API Reference

The core of the library is the `StockDataProcessor` class. All methods are static for convenience, allowing use without instantiation. Below is a detailed breakdown of each method, including parameters, returns, examples, and edge cases.

`__init__(self, df=None)`

Initializes the processor with an optional DataFrame. Useful for instance-based workflows (e.g., chaining operations).

- **Parameters**:
  - `df` (pd.DataFrame, optional): Input DataFrame to process.
- **Returns**: None.
- **Example**:

  ```python
  processor = StockDataProcessor(df)
  # Access via instance if needed, though static methods are preferred.
  ```

`download_stock_data(ticker, period="1y", interval="1d")` (Static)

Downloads historical OHLCV data from Yahoo Finance.

Parameters

- `ticker` (str): Stock symbol (e.g., "AAPL").
- `period` (str): Time span (e.g., "1y", "max"). See yfinance docs for options.
- `interval` (str): Data frequency (e.g., "1d", "1h").

Returns

- `pd.DataFrame` : Columns: `date`, `open`, `high`, `low`, `close`, `volume` .

Raises

- `ValueError` : Invalid ticker or download failure.

Edge Cases

- Multi-index columns are flattened.
- Empty results return an empty DataFrame.

Example

```python
df = StockDataProcessor.download_stock_data("TSLA", period="3mo", interval="1d")
print(df.head())  # Displays recent data
```

`fill_missing_dates(df, date_col='date', break_date=False)` (Static)

Fills gaps in dates using business day offsets and optionally extracts date components.

Parameters

- `df` (pd.DataFrame): Input with date column.
- `date_col` (str): Name of date column.
- `break_date` (bool): If True, adds `day`, `month`, `year` columns.

Returns

- `pd.DataFrame`: Filled DataFrame.

Algorithm

- Sorts by index.
- For NaN dates: Uses previous/next valid date + BDay offset.
- Fallback: Current timestamp if isolated.

Example

```
df['date'] = pd.to_datetime(df['date'])  # Ensure datetime
df_filled = StockDataProcessor.fill_missing_dates(df, break_date=True)
print(df_filled[['date', 'day', 'month', 'year']].head())
```

`markov_impute(series, n_bins=20, strategy="mode")` (Static, Internal)

Markov chain-based imputation for series (used in `fill_nan_advanced`).

Parameters

- `series` (pd.Series): Input series.
- `n_bins` (int): Number of states.
- `strategy` (str): "mode" (deterministic) or "random" (stochastic).

Returns

- `pd.Series`: Imputed series.

Algorithm

- Bins non-NaN values.
- Builds transition matrix.
- Predicts next state from previous.

Example

```
imputed = StockDataProcessor.markov_impute(df['close'].dropna())
```

`fill_nan_advanced(df, col_tech_map)` (Static)

Applies sequential imputation techniques per column.

Parameters

- `df` (pd.DataFrame): Input DataFrame.
- `col_tech_map` (dict): {col: [(tech, params)]}. See method docstring for full list (e.g., "knn", "kalman").

Returns

- `pd.DataFrame`: Filled copy.

Supported Techniques

Technique	Description	Params Example
`drop`	Drop rows with NaNs in col	`{}`

Technique	Description	Params Example
`mean` / `median` / `mode`	Statistical fill	`{}`
`ffill` / `bfill`	Forward/backward fill	`{}`
`sma` / `rolling`	Simple moving average	`{'window': 14}`
`ema`	Exponential moving average	`{'alpha': 0.3}`
`linear` / `quadratic` / `cubic`	Polynomial interpolation	`{}`
`knn`	K-Nearest Neighbors	`{'n_neighbors': 3}`
`markov`	Markov chain	`{}`
`weighted_combo`	0.5SMA + 0.5EMA	`{'window': 14, 'alpha': 0.3}`
`kalman`	Kalman smoothing	`{}`

Raises

- `ValueError` : Unknown technique.

Example

```
col_tech_map = {
    "close": [("knn", {"n_neighbors": 5}), ("kalman", {})],
    "volume": [("sma", {"window": 10})]
}
df_filled = StockDataProcessor.fill_nan_advanced(df, col_tech_map)
```

`plot_graph(df, columns, graph_type, size=(10,6), color='blue', stacked=False)` (Static)

Generates diverse plots for EDA.

Parameters

- `df` (pd.DataFrame): Input data.
- `columns` (list[str]): Columns to plot (varies by type).
- `graph_type` (str): See table below.
- `size` (tuple): Figure size.
- `color` (str/list): Color scheme.
- `stacked` (bool): For area plots.

Supported Graph Types

Type	Columns Req.	Library	Use Case
`line` / `scatter`	2 (x,y numeric)	Matplotlib	Trends/Correlations
`bar`	1-2	Matplotlib	Counts/Categories
`hist`	1 (numeric)	Matplotlib	Distributions
`box` / `violin`	1 (numeric)	Seaborn	Outliers/Summary
`pairplot`	>=2	Seaborn	Multi-var EDA
`area` / `stacked_area` / `stream`	>=2	Matplotlib	Cumulative Trends
`pie`	1-2	Matplotlib	Proportions
`waterfall`	2	Matplotlib	Cumulative Changes
`candlestick`	4+ OHLC	mplfinance	Price Action

Type	Columns Req.	Library	Use Case
`treemap` / `sunburst`	2+	Plotly	Hierarchies
`choropleth`	2 (region,value)	Plotly	Geo-Maps

Returns

- None (displays plot).

Raises

- `ValueError` : Invalid columns/type.

Example

```
StockDataProcessor.plot_graph(df, ["date", "close"], "line")
StockDataProcessor.plot_graph(df, ["open", "high", "low", "close"], "candlestick", size=(12,8))
```

`detect_outliers_advanced(df, numeric_cols, z_thresh=3, mod_z_thresh=3.5, rolling_window=5, price_change_thresh=0.05, plot_graphs=True, combine='union', vote_thresh=None)` (Static)

Detects outliers using 7 methods.

Parameters

- `df` (pd.DataFrame): Input.
- `numeric_cols` (list[str]): Columns to analyze.
- Detection thresholds as named.
- `combine` (str): "union"/"intersection".
- `vote_thresh` (int): Min methods to flag (overrides combine).
- `plot_graphs` (bool): Generate plots (box, hist, scatter, violin).

Detection Methods

Method	Description	Threshold
`z_score`		Z
`modified_z`	Robust Z via MAD	Median ± MAD
`iqr`	Outside 1.5*IQR	Q1-Q3
`rolling`	Outside rolling mean ± 3*std	Window-based
`price_change`		% change
`returns_z`	Z on returns	Returns
`cusum`	Cumulative shifts	>3*std

Returns

- dict: {col: {'per_method': {method: set(indices)}, 'combined': set(indices)}}.

Example

```
outliers = StockDataProcessor.detect_outliers_advanced(df, ["close"], plot_graphs=True, combine="intersection")
print(f"Outliers in close: {len(outliers['close']['combined'])}")
```

`treat_outliers(df, outlier_results, treatment_map)` (Static)

Applies treatments to detected outliers.

Parameters

- `df` (pd.DataFrame): Input.
- `outlier_results` (dict): From `detect_outliers_advanced`.
- `treatment_map` (dict): {col: [(method, params)]}.

Supported Treatments

Method	Description	Params Example
`delete`	Drop outlier rows	`{}`
`winsorize` / `cap`	Clip to quantiles	`{'lower':0.01, 'upper':0.99}`
`median_replace`	Replace with median	`{}`
`mean_cap`	Clip to mean ± k*std	`{'k':3}`
`log_transform` / `sqrt_transform` / `boxcox`	Transformations	`{}` (clip negatives)
`robust_flag`	Add flag column	`{}`
`interpolate_linear` etc.	Interpolate	`{'method':'linear'}`
`rolling_mean` / `rolling_median`	Smoothing	`{'window':5}`
`ema_smooth`	EMA	`{'alpha':0.3}`
`kalman`	Kalman filter	`{}`
`markov_prev` / `markov_avg`	Markov replacements	`{}`

Returns

- `pd.DataFrame`: Treated copy.

Raises

- `ValueError`: Unknown method.

Example

```
treatment_map = {"close": [("winsorize", {"lower":0.05, "upper":0.95}), ("robust_flag", {})]}
df_treated = StockDataProcessor.treat_outliers(df, outliers, treatment_map)
```

`test_data()` (Static)

Generates sample OHLCV DataFrame (20 business days, Oct 2025).

Returns

- `pd.DataFrame`: Columns: `date`, `low`, `high`, `open`, `close`, `volume`.

Example

```
test_df = StockDataProcessor.test_data()
print(test_df.describe())
```

Usage Examples: End-to-End Workflow

1. Basic Data Pipeline

```
# Load test data
df = StockDataProcessor.test_data()

# Introduce NaNs for demo
df.loc[5:7, "close"] = np.nan

# Fill dates (already complete in test)
df = StockDataProcessor.fill_missing_dates(df)

# Impute NaNs
col_tech_map = {"close": [("linear", {})]}
df = StockDataProcessor.fill_nan_advanced(df, col_tech_map)

# Visualize
StockDataProcessor.plot_graph(df, ["date", "close"], "line")

# Outliers
outliers = StockDataProcessor.detect_outliers_advanced(df, ["volume"], price_change_thresh=0.1)
print(outliers)
```

2. Real-World: AAPL Analysis with Outliers

```
df = StockDataProcessor.download_stock_data("AAPL", "1y")
df = StockDataProcessor.fill_missing_dates(df)

# Detect outliers in returns
df["returns"] = df["close"].pct_change()
outliers = StockDataProcessor.detect_outliers_advanced(
    df, ["returns"], z_thresh=2.5, plot_graphs=False, vote_thresh=3
)

# Treat: Smooth returns
treatment_map = {"returns": [("ema_smooth", {"alpha": 0.2})]}
df_treated = StockDataProcessor.treat_outliers(df, outliers, treatment_map)

# Plot treated data
StockDataProcessor.plot_graph(df_treated, ["date", "returns"], "scatter", color="red")
```

3. Advanced Imputation Chain

```
col_tech_map = {
    "open": [("ffill", {}), ("knn", {"n_neighbors": 4})],
    "volume": [("markov", {}), ("kalman", {})]
}
df_imputed = StockDataProcessor.fill_nan_advanced(df, col_tech_map)
StockDataProcessor.plot_graph(df_imputed, ["date", "volume"], "area", stacked=True)
```

Performance Considerations

- **Scalability**: Handles up to 10k rows efficiently; for larger datasets, subsample or parallelize imputations.
- **Memory**: Copies DataFrames; use `inplace=True` where possible in custom extensions.
- **Compute-Intensive Methods**: KNN/Kalman scale O(n^2)/O(n); limit to key columns.

Contributing

1. Fork the repo.
2. Create a feature branch (`git checkout -b feature/amazing-feature`).
3. Commit changes (`git commit -m 'Add amazing feature'`).
4. Push and open a PR.

Run tests: `pytest tests/` . Lint: `black .` .

Acknowledgments

- Built with love by [QUANT-FINANCE COMMUNITY , NIT ROURKELA].
- Thanks to yfinance AND the open-source community.

For issues or features, open a GitHub issue. Let's make stock analysis accessible!