Task – 3

Topic- High- performance Time series Transformation

Created by- Kirti Bala

Overview of High- performance time series Transformation with NumPy & Pandas

Objective

- Efficiently compute common time series transformations on large datasets (>1 million rows):
- Rolling statistics (mean, variance) – large windows, millions of rows.
- Exponentially weighted moving averages (EWMA) and covariances – memory efficient.
- FFT-based spectral analysis & band-pass filtering- frequency domain insights.

➢ Challenges

- Large data = memory pressure
- Python loops = too slow; prefer vectorization

- Need benchmarking: NumPy vs pandas vs optimized (Numba/ stride tricks)

➢ Constraints:

Handle >1M rows (simulate large sensor data).

Compare pure NumPy vs pandas built-ins vs accelerated (Numba/stride tricks).

Benchmark runtime + memory.

Auto-select fastest method based on dataset size.

➢ Deliverables:

✓timeseries_utils.py → Implementations.
✓benchmark.py → Benchmarking scripts.
✓results.csv + Report summarizing findings.

2. Directory Structure

```
high_perf_timeseries

---- timeseries_utils.py    # Core functions
———— benchmark.py           # Benchmarking
code
———— results/
---- benchmark_results.csv
  └── plots/
---- report.md              # Performance report
└── requirements.txt
```

3. Implementation Details

3.1 Rolling Window Statistics

3.1.1 pandas Implementation

```
import pandas as pd
```

```
def rolling_pandas(df: pd.DataFrame, window: int):
    """Rolling mean & variance using pandas built-ins."""
    rolling_mean = df.rolling(window).mean()
    rolling_var = df.rolling(window).var()
    return rolling_mean, rolling_var
```

3.1.2 NumPy Implementation (Cumulative Sum Trick)

```
import numpy as np

def rolling_numpy(arr: np.ndarray, window: int):
    """Rolling mean & variance using cumulative sums (pure NumPy)."""
    cumsum = np.cumsum(np.insert(arr, 0, 0, axis=0), axis=0)
    cumsum_sq = np.cumsum(np.insert(arr**2, 0, 0, axis=0), axis=0)
```

```
    mean = (cumsum[window:] - cumsum[:-window]) / window
    var = ((cumsum_sq[window:] - cumsum_sq[:-window]) / window) - mean**2
    return mean, var
```

3.1.3 Accelerated with Numba

```
from numba import njit

def rolling_numba(arr, window):
    n, m = arr.shape
    means = np.empty((n - window + 1, m))
    vars_ = np.empty((n - window + 1, m))

    for j in range(m):
        for i in range(n - window + 1):
            window_data = arr[i:i+window, j]
```

```
        mean = np.mean(window_data)
        means[i, j] = mean
        vars_[i, j] = np.var(window_data)
    return means, vars_
```

3.2 EWMA & Covariance

1. pandas EWMA

```
def ewma_pandas(df: pd.DataFrame, span: int):
    return df.ewm(span=span, adjust=False).mean()


def ewm_cov_pandas(df1: pd.DataFrame, df2: pd.DataFrame, span: int):
    return df1.ewm(span=span).cov(df2)
```

2. NumPy EWMA (Vectorized)

```
def ewma_numpy(arr: np.ndarray, alpha: float):
    """Compute EWMA for each column using NumPy."""
    out = np.zeros_like(arr)
    out[0] = arr[0]
    for i in range(1, arr.shape[0]):
        out[i] = alpha * arr[i] + (1 - alpha) * out[i-1]
    return out
```

3. Covariance via EWMA

```
def ewm_cov_numpy(arr1: np.ndarray, arr2: np.ndarray, alpha: float):
    mean1 = ewma_numpy(arr1, alpha)
  mean2 = ewma_numpy(arr2, alpha)
    cov = ewma_numpy((arr1 - mean1) * (arr2 - mean2), alpha)
    return cov
```

3.3 FFT & Band-Pass Filter

3.3.1 FFT Spectral Analysis

```
from scipy.fft import fft, fftfreq, ifft

def spectral_analysis(arr: np.ndarray, sampling_rate: float):
    n = arr.shape[0]
    freqs = fftfreq(n, d=1/sampling_rate)
    spectrum = fft(arr, axis=0)
    return freqs, spectrum
```

3.3.2 Band-Pass Filtering

```
def bandpass_filter(arr: np.ndarray, low: float, high: float, sampling_rate: float):
```

```
    freqs, spectrum = spectral_analysis(arr, sampling_rate)
    mask = (np.abs(freqs) >= low) & (np.abs(freqs) <= high)
    filtered_spectrum = spectrum * mask[:, None]
    filtered_signal = np.real(ifft(filtered_spectrum, axis=0))
    return filtered_signal
```

4. Auto-Select Fastest Method

```
def auto_select_rolling(arr: np.ndarray, window: int):
    n = arr.shape[0]
    if n > 2_000_000:
        return rolling_numba(arr, window)  # Numba best for huge data
    elif n > 500_000:
```

        return rolling_numpy(arr, window) # NumPy for mid-scale

    else:

        df = pd.DataFrame(arr)

        return rolling_pandas(df, window) # pandas for small

---

5. Benchmarking (benchmark.py)

5.1 Setup

import time

import pandas as pd

import numpy as np

import psutil

import matplotlib.pyplot as plt

```
from timeseries_utils import (
    rolling_numpy, rolling_pandas,
rolling_numba,
    ewma_numpy, ewma_pandas,
spectral_analysis
)


def memory_usage():
    return psutil.Process().memory_info().rss /
1e6  # MB
```

5.2 Benchmark Function

```
def benchmark_methods():
    sizes = [100_000, 500_000, 1_000_000,
2_000_000]
    window = 100
    results = []

    for n in sizes:
```

```
    data = np.random.randn(n, 3)  # 3 features
    df = pd.DataFrame(data)

    # pandas rolling
    start, mem_start = time.time(),
memory_usage()
    rolling_pandas(df, window)
    results.append(("pandas", n, time.time()-
start, memory_usage()-mem_start))

    # numpy rolling
    start, mem_start = time.time(),
memory_usage()
    rolling_numpy(data, window)
    results.append(("numpy", n, time.time()-
start, memory_usage()-mem_start))

    # numba rolling
    start, mem_start = time.time(),
memory_usage()
```

```
      rolling_numba(data, window)
      results.append(("numba", n, time.time()-start, memory_usage()-mem_start))

   df_results = pd.DataFrame(results, columns=["method", "size", "time", "memory"])

df_results.to_csv("results/benchmark_results.csv", index=False)

   # Plot
   for metric in ["time", "memory"]:
      for method in df_results["method"].unique():
         subset = df_results[df_results["method"] == method]
         plt.plot(subset["size"], subset[metric], label=method)
      plt.xlabel("Size")
      plt.ylabel(metric.capitalize())
```

```
    plt.legend()
    plt.title(f"{metric.capitalize()} Benchmark")
    plt.save
fig(f"results/plots/{metric}_benchmark.png")
    plt.clf()
```

Run via:

python benchmark.py

6. Report (report.md)

Structure:

6.a.Overview

Problem, dataset size, requirements.

6.b. Methods Compared

- pandas, NumPy, Numba.

6.c. Performance Tables

- Runtime & memory per method.

6.d. Plots

- Time vs dataset size.
- Memory vs dataset size.

6.e. Recommendations

- pandas good for ≤500k rows.
- NumPy good for 0.5M–2M rows.
- Numba best for >2M rows.

7. requirements.txt

numpy

pandas

matplotlib

numba

psutil

scipy