scMSI Testing Documentation

Version: 1.0
Date Compiled: September 2024

Overview

This testing documentation outlines rigorous testing procedures for scMSI, a bioinformatics tool for deconvoluting microsatellite length distributions. These procedures are intended for bioinformatics professionals experienced in large-scale data processing and high-performance computing environments.

Testing Prerequisites

- Python 3.7 environment with HPC-optimized libraries (Intel MKL, OpenBLAS)
- Minimum of 512 GB RAM and 64 CPU cores
- Availability of bwa, samtools, bedtools, gurobi, and the scMSI codebase
- Access to example genomic data and test files in the scMSI repository

Test Data Specification

Test data is available within the scMSI repository under the `Examples/` directory. The following commands can be used to download additional high-complexity test datasets:

```
wget https://github.com/SeqAnalysis/scMSI.git
unzip scMSI-main.zip
```

Test Command Execution

Run the following commands to test the parallelized execution of scMSI using multiple threads and high-performance I/O configurations:

Run main.py:

Run the following command to fit the scMSI model using the specified tumor data (Examples/ tumor_0.5.txt) and control data (Examples/ tum_0.5.txt). The results will be saved in the results/ directory. The program utilizes 64 threads for parallel execution and includes debug-level logging for detailed error tracking:

python main.py -d Examples/tumor_data.txt -c Examples/control_data.txt -o results/ --threads 64 --log-level DEBUG

Upon completion, main.py will output the mean values of each component (means) and the mixing proportions of different components (Mixed proportion). Based on the Mixed proportion, main.py will select the appropriate means value to be used as the value parameter in Z_test.py for the z-test.

Run Z_test.py:

Run the following command to perform the z-test using the selected mean value from main.py as the value parameter:

python Z_test.py

Z_test.py will use the selected means value from main.py as the value parameter in the z-test calculation. The output of Z_test.py will indicate the Microsatellite status.In Z_test.py, the sw.ztest function is used to perform a one-sample z-test. The function compares the mean of the sample data (X1) against a hypothesized population mean specified by the value parameter. In this case, the code uses:

sw.ztest(X1, value=13.69)

Here, X1 is the normal sample data loaded from the file, and value is the hypothesized mean (in this case, 13.69). The function returns a tuple containing:

The z-statistic, which measures the difference between the sample mean and the hypothesized mean in terms of standard errors.

The p-value, which indicates the probability of observing a z-statistic as extreme as the one calculated, assuming the null hypothesis (that the true mean is 13.69) is true.

Z_test.py uses the p-value from the ztest result to determine the Microsatellite status:

If the p-value is greater than or equal to 0.001, the output will be Microsatellite status: mss.

If the p-value is less than 0.001, the output will be Microsatellite status: msi.

In this workflow, the results of main.py directly determine the value used in Z_test.py, ultimately influencing the final Microsatellite status classification.

Output File Format and Validation

The output will include:
- `means `: Mean estimates for each sub-clone.
- `covariances `: Covariance matrices for the sub-clonal distributions.
- `fractions `: Sub-clonal fractions.

Microsatellite status:mss

or

Microsatellite status:msi

Validate the output by comparing it to reference results in the `expectedResults_large/` directory, using checksum validation:
```
```

sha256sum results/* expectedResults_large/*
```

## Performance Profiling and Optimization

Use performance profiling tools such as `gprof` or Intel VTune to analyze CPU-bound or memory-bound performance bottlenecks:

```
gprof python main.py gmon.out > profiling_results.txt
```

Alternatively, run VTune to profile parallel performance across multiple cores:

```
vtune -collect hotspots -- python main.py -d Examples/tumor_data.txt -c Examples/control_data.txt -o results/ --threads 64
```

Troubleshooting

- If execution fails due to memory exhaustion, reduce the number of parallel threads and monitor memory usage with:

```
htop
```

- For numerical stability issues (e.g., NaN values in the covariance matrix), ensure that input data is properly normalized. Use the `numpy` function:

```
X = (X - X.mean(axis=0)) / X.std(axis=0)
```
```