GSoC 2025 - ruptures tests

Minh Long Nguyen

2025-04-04

This document presents the solutions to the tests provided in the ruptures project, part of the Google Summer of Code program for 2025. The working files, including both R and C++ scripts, are included in the accompanying folder and should be regarded as supplementary materials.

For the sake of this work, we count from 0, not 1.

Easy test

In this task, we are given a 2D array X of size $T \times D$ and are expected to produce a cumulative sum matrix (by row) named Y, such that:

1. $Y_0 = 0^D$
2. $Y_i = \sum_{j<i} Y_j, \ i = 1, ..., D$

Cpp code We first initialize a cumulative sum (cumsum) matrix of size $(T+1) \times D$, filled with zeros. Then, we iteratively compute cumulative sums, reusing the output from the previous iteration to improve efficiency.

The C++ solution is given below.

```
arma::mat Cpp_getCumsum(const arma::mat& X) {

  int nr = X.n_rows;
  int nc = X.n_cols;

  arma::mat CumsumMat(nr+1, nc, arma::fill::zeros);

  for(int i=0; i<nr; i++) {

    CumsumMat.row(i+1) = CumsumMat.row(i) + X.row(i);
    //use output from previous iteration for efficiency

  }
  return CumsumMat;
}
```

Testing and benchmarking The function is wrapped using RcppArmadillo for interfacing with R. A generally vectorised R function, R_getCumsum(), that does the same computation is prepared for comparision purposes.

We simulate an 1000×100 matrix X whose values come from i.i.d. standard Gaussian distributions.

```r
set.seed(1)
X = matrix(rnorm(10^5), nrow = 1000)
```

The outputs of Cpp_getCumsum() and R_getCumsum() on X are compared.

```r
all.equal(R_getCumsum(X), Cpp_getCumsum(X))
```

```
## [1] TRUE
```

Good news! They are equal. What about computational cost?

We can use the microbenchmark package in R to compare the runtimes.

```r
microbenchmark(R_getCumsum = R_getCumsum(X),
               Cpp_getCumsum = Cpp_getCumsum(X))
```

```
## Unit: microseconds
##           expr   min      lq     mean  median     uq    max neval
##    R_getCumsum 961.5 1493.75 2279.335 1906.35 2273.4 8191.5   100
##  Cpp_getCumsum 171.8  289.40  533.809  338.85  592.5 3757.0   100
```

Cpp_getCumsum() is 4-6 times faster than R_getCumsum().

Medium test

In this task, we are required to compute this function $f(i', j)$, where i' is the starting index, j is the ending index, and

$$f(i', j) = \sum_{i=i'}^{j-1} \left|X_i - m\right|^2,$$

where $|\cdot|$ is the Euclidean norm operator, and m is the centroid of the set $\{X_k\}_{k \in i':(j-1)}$. This can be viewed as the total squared Euclidean distance between all data points in a cluster and its centroid m (k-means clustering!!!).

A simple approach to compute $f(i', j)$ is as follows:

1. Compute the centroid m of $\{X_k\}_{k \in i':(j+1)}$.
2. Calculate the squared Euclidean distance from each data point to m and return the total.

The following Cpp program implements this approach. We create a class **Cost** operating on arma::mat objects as required by the task. The method eval() compute $f(i', j)$.

```cpp
class Cost {
private:
  arma::mat X;

public:
```

```
  Cost(const arma::mat& inputMat) { //initialise a Cost object
    X = inputMat;
  }


  double Cpp_Eval(int start, int end) const {  //no precomputation

    int nc = X.n_cols;
    int ncr = end - start;
    arma::rowvec sumX =  arma::zeros<arma::rowvec>(nc);

    for(int i=start; i<end; i++) {
      sumX = sumX + X.row(i);
    }

    arma::rowvec meanX = sumX/ncr;

    double error = 0;
    double eucldist;
    for(int i=start; i<end; i++) {
      eucldist = arma::norm(X.row(i) - meanX, 2);
      error = error + std::pow(eucldist, 2);
    }

    return error;
  }

};
```

RCPP_MODULE() allows us to use the class Cost in R with object Cpp_Eval.

```
RCPP_MODULE(mod_Cost) {
  class_<Cost>( "Cost")
  .constructor<arma::mat>()
  .method( "Cpp_Eval", &Cost::Cpp_Eval)
  ;
}
```

Testing We then create a generally vectorised R function called R_eval() that does the same computation for comparision purposes.

```
R_eval = function(X, start, end){
  R_start = start+1
  R_end = end

  Xe = X[R_start:R_end,]
  cMXe = colMeans(Xe)

  return(sum(sweep(Xe, 2, cMXe, FUN = "-")^2))
}
```

Let's create an object and compare the results by trying to calculate the distance from all data points in the matrix to the centroid

```r
Xnew = new(Cost, X) #Cpp object
all.equal(R_eval(X,0, 1000), Xnew$Cpp_Eval(0, 1000))
```

```
## [1] TRUE
```

Good news! They give equal results. What about runtimes? We will come back to that later.

Hard test

Our task is to compute $f(i', j)$ in constant time.

Is that even possible?

Each k-means iteration would be done in $\mathcal{O}(k)$ times. That's great news.

. . . Yes and no. . .

Yes, we can pre-compute some quantities and then easily compute $f(i', j)$ in constant time.

And no, unfortunately, these quantities are computed in linear time.

There is no free lunch.

We won't go into much detail about these pre-computed quantities as they have been described in this GSoC wikipage..

The computations described in this are similar to this well-known equation:

$$\mathbb{V}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

In theory if you know $\mathbb{E}[X^2]$ and $\mathbb{E}[X]$, then you also know $\mathbb{V}[X]$.

Obviously, we don't know these quantities in most applications (except for toy examples).

We can more or less do the same thing here, i.e., pre-computing quantities sharing similar information as $\mathbb{E}[X^2]$ and $\mathbb{E}[X]$. These are getCumsum$(X_{0:T})$ and getCumsum$(X^2_{0:T})$. These can be pre-computed and saved as attributes of a \textbf{Cost object.

That results in the Cpp_effEval() function, which is described in the following Cpp program, which also contains solutions to the previous two tests, as we merge them into a single Cpp file.

```cpp
// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]]
arma::mat Cpp_getCumsum(const arma::mat& X) {

  int nr = X.n_rows;
  int nc = X.n_cols;

  arma::mat CumsumMat(nr+1, nc, arma::fill::zeros);

  for(int i=0; i<nr; i++) {
```

```
    CumsumMat.row(i+1) = CumsumMat.row(i) + X.row(i);
    //use output from previous iteration for efficiency

  }

  return CumsumMat;

}

class Cost {
private:
  arma::mat X;
  arma::mat CSX; //cumsum(X)
  arma::mat CSXsq; //cumsum(Xsq)

public:

  Cost(const arma::mat& inputMat) { //initialise a Cost object
    X = inputMat;
    CSX = Cpp_getCumsum(inputMat);
    CSXsq = Cpp_getCumsum(arma::pow(inputMat, 2));
  }


  double Cpp_Eval(int start, int end) const {  //no precomputation

    int nc = X.n_cols;
    int ncr = end - start;
    arma::rowvec sumX =  arma::zeros<arma::rowvec>(nc);

    for(int i=start; i<end; i++) {
      sumX = sumX + X.row(i);
    }

    arma::rowvec meanX = sumX/ncr;

    double error = 0;
    double eucldist;
    for(int i=start; i<end; i++) {
      eucldist = arma::norm(X.row(i) - meanX, 2);
      error = error + std::pow(eucldist, 2);
    }

    return error;
  }


  double Cpp_effEval(int start, int end) const {  //use precomputation

    int ncr = end - start;
    double errsumXsq =  arma::sum(CSXsq.row(end) - CSXsq.row(start));
    double sqerrsumX =  std::pow(arma::norm(CSX.row(end) - CSX.row(start),2), 2);
```

```
    return errsumXsq - sqerrsumX/ncr;

  }

};

RCPP_MODULE(mod_Cost) {  //to use the class Cost in R
  class_<Cost>( "Cost")
  .constructor<arma::mat>()
  .method( "Cpp_Eval", &Cost::Cpp_Eval)
  .method( "Cpp_effEval", &Cost::Cpp_effEval)
  ;
}
```

Cpp code - final

Testing and benchmarking We can check whether or not Cpp_effEval() give the same results as R_eval() and Cpp_Eval().

```
Xnew = new(Cost, X)
all.equal(R_eval(X, 0, 1000), Xnew$Cpp_Eval(0, 1000), Xnew$Cpp_effEval(0, 1000))
```

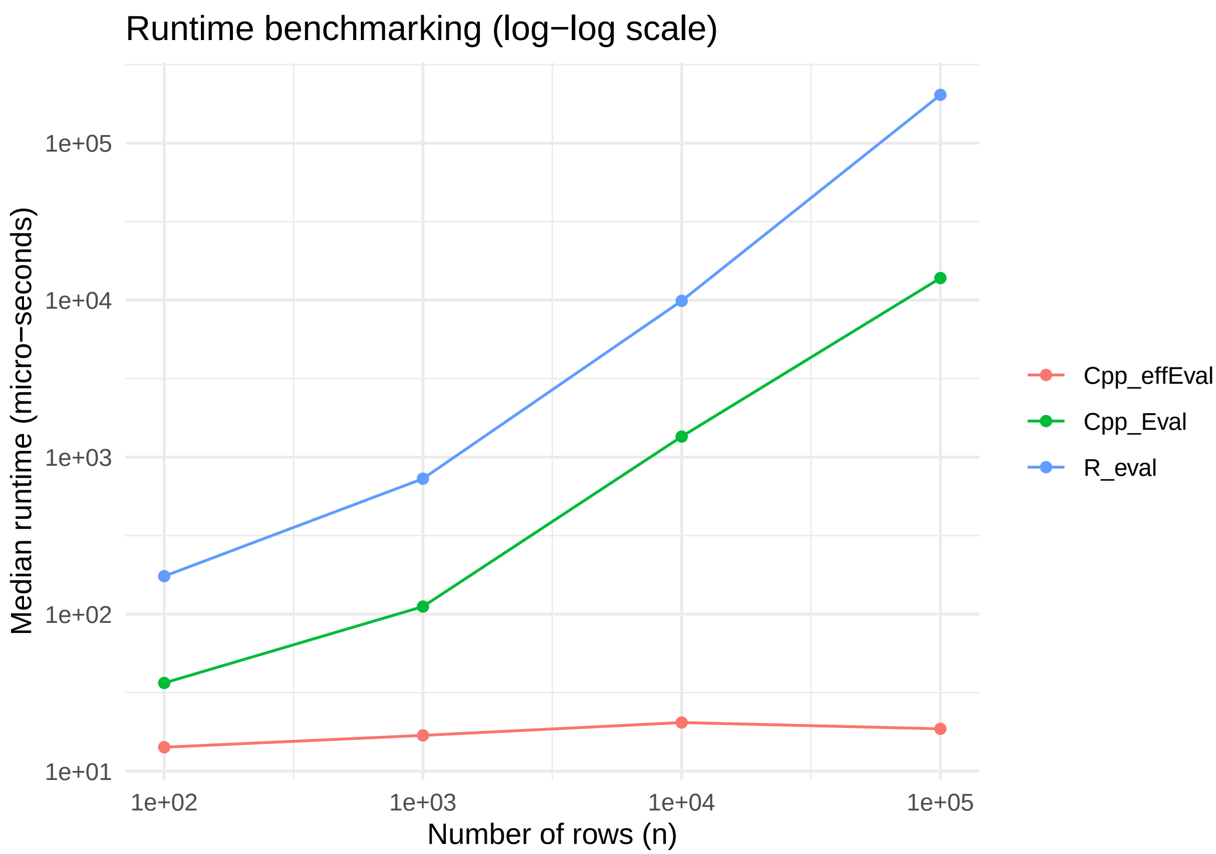
```
## [1] TRUE
```

Great news! The results are the same.

Benchmarking We will vary the number of rows $n \in \{1000, 10000, 10000, 100000\}$. These datasets are simulated using the same methodology where the number of columns is 100.

```
runtime = read.csv("./runtime.csv") #load the benchmarking results (see runtime_eval())

ggplot(runtime, aes(x = n, y = runtime, group = method, color = method)) +
  geom_line() +
  geom_point() +
  scale_x_log10() +
  scale_y_log10() +
  labs(
    title = "Runtime benchmarking (log-log scale)",
    x = "Number of rows (n)",
    y = "Median runtime (micro-seconds)"
  ) +
  theme_minimal() +
  theme(legend.title = element_blank())
```



We can conclude that our efficient effEval function achieves a constant runtime complexity as the log-log plot of median runtime vs. n appears to be a near-constant line.

However,

there is no free lunch.

No one will give us $\text{getCumsum}(X_{0:T})$ and $\text{getCumsum}(X_{0:T}^2)$ for free.

The design of the Cost class is mostly useful in the scenarios where multiple computations rely on these pre-computed matrices.

This's the end of my work. Thanks for your time!