SPRT Documentation

Weicong Chen

June 16, 2020

Abstract

Sequential Probability Ratio Test (SPRT) statistics are performed under the context of General Linear Model (GLM). These statistics make dynamic decisions on activation status and stopping of administration of a difficulty level, which is switched progressively based on observed blood-oxygen-level dependent (BOLD) signals. A computational challenge in real-time neurocomputing with fMRI is that there are literally 100,000s of potential models at the voxel that must be processed and analyzed simultaneously. As whole-brain fMRI data is acquired in 3 second windows, efficiency and parallelization are essential for system performance. To fully accommodate the capability of real-time processing and result feedback, SPRT analysis is implemented using state of the art parallel computing techniques with deep optimization.

1 Development Tools

We decided to use Intel Parallel Studio as the general development platform since it will unleash the performance of Intel Xeon processor. Intel Cilk Plus library is used to implement OpenMP-based parallelism for multi-core, manycore and coprocessors. Intel Math Kernel Library help speed math processing in scientific, engineering and financial applications. It also provides functionality for dense and sparse linear algebra (BLAS, LAPACK, PARDISO), FFTs, vector math, summary statistics and more.

2 Real-time Collected Data

n: number of scans
j: voxel number
k: number of covariants

In our current real-subject experiments, $n = 238$, total # of $j = 36 * 128 * 128$ (each scan we get a scan of full brain volume, which constructs a 3-d matrix of data points), $k = 8$.

For each voxel:
$X : n * k$
$Y : n * 1$

Note:
Y is known as the Blood Oxygen Level Dependent (BOLD) response. It is recorded from the fMRI scanner.
X is the pre-tuned design matrix to coordinate with the given tasks.
For each voxel, we create a $n * 1$ vector to store BOLD response, and incrementally record data points from the scanner as the experiment progresses.

3 Formulas

3.1 GLM

Under the General Linear Model (GLM) settings, we estimate model parameter β in the following way:

$Y = X\hat{\beta} + \epsilon$

where $\hat{\beta}$ is the estimates of the slope in GLM model, ϵ is the error term

We hence estimate $\hat{\beta}$ as follow:

$\hat{\beta} = (X'X)^{-1}X'Y$

where $\hat{\beta} : k * 1$

3.2 Hubor (Sandwich) Estimator

Hubor Estimator:

$H = X(X'X)^{-1}X'$

$H : n * n$

Define diagnal elements h_{ii} where $i = 1 \ldots n$, which form the unbiased sandwitch estimator.

Define D_r^* is a $n * n$ matrix contain only dignal elements r_{ii} where $r_{ii} = \dfrac{Y_i - X\hat{\beta}}{1 - h_{ii}}$

Optimization note:
Intel MKL Sparse Matrix Vector Multiply Prototype Package is implemented to improve sparse matrix-vector multiply for Intel Xeon processor. A 2-stage API is used for sparse BLAS (analyze and execute). Performance improvement: 2 - 200x depending on matrix sparsity and n.

3.3 SPRT

$$\mathrm{Var}[c\hat{\beta}] = c'(X'X)^{-1}X'D_r^*X(X'X)^{-1}c$$

$$Z = \frac{c^T\hat{\beta}}{\sqrt{\mathrm{Var}[c^T\hat{\beta}]}}$$

$$\mathrm{SPRT} = [(c\hat{\beta} - \theta_0)^T * \mathrm{Var}[c\hat{\beta}]^{-1} * (c\hat{\beta} - \theta_0) - (c\hat{\beta} - \theta_1)^T * \mathrm{Var}[c\hat{\beta}]^{-1} * (c\hat{\beta} - \theta_1)]/2$$

Note:
Voxel-wise SPRT value is computed in a parallel computing manner using the classic map-reduce processing model using Intel Cilk Plus routines. Our 16-core Intel Xeon processor shows speed improvment of about 12 times faster than serial computation.
Once we get estimate after K scans, let's compute a target θ_1 for each voxel in each contrast. We only set θ_1 once, for each voxel. Z is a universal value across the brain. Default Z: 3.12.

$$\theta_1 = Z * \sqrt{\mathrm{Var}[c\hat{\beta}]}$$

SPRT boundary. Default α: 0.001. Default β: 0.1.

$$\text{upper bound: } \frac{log(1-\beta)}{\alpha}$$
$$\text{lower bound: } \frac{log(1-\alpha)}{\beta}$$

Stopping Rule: (Current version as of June 16, 2020) If a total of 90% of voxels has SPRT value greater than the upper bound and smaller than the lower bound. A stopping is called.

4 Using Spark

4.1 Idea

- master node performs all the pre-processing (Data initialization, IO, checking, etc.)
- Voxel-specific data: Y and D_r^*, along with voxel coordinates x, y and z are packed into custom class `DistributedDataset` and transform to Spark RDD.and send to Spark executors.
- Scan-wise Shared Variables: design matrix X, Hubor Estimator H, etc, are broadcasted using `SparkContext.broadcast()` .
- To save bandwidth, only newly recorded BOLD responses are send to executors. Old RDDs are persisted inside executor's memory. New RDD are generated on executors side using the old RDDs. (Not sure if it does speed up.)
- Each executor performs voxel-wise matrix computation to generate results such as Z score, variance, SPRT value, activation stats etc, and packed into another custom class `CollectedDataset` and will be collected by the master.

- Matrix computation (Multiply, sparse multiply, inverse) on executor utilize Intel MKL by calling native methods using Java Native Interface (JNI).
- Master node peforms post-processing to the received `List<CollectedDataset>`.

4.2 Future Work and Thoughts

- Minimizing data flow?
- Look into Spark Streaming?