Median of Means for an AR(1) | Christelle Xu

Summary

This is an illustration of my current progress of the Median of Means (MOM) algorithm implemented for an AR(1). At this point in time, the algorithm does not work correctly though any issues will be resolved by graduation. The purpose of this document is to present the algorithm, the progress made, and the problems to be solved.

Median of Means (MOM) is a robust alternative to the Empirical Risk Minimization algorithm (ERM), that aims to tackle computational complexity and is a way to design and analyze efficient, parallel, and robust algorithms for learning. The ERM in and of itself has benefited from regularization given its propensity to overfit, but has also been found to suffer in terms of performance upon deviating from iid subgaussian setups such as when the data is heavy-tailed.

The MOM algorithm is one important solution to this issue particularly in its capacity to handle large amounts of data. Its primary advantage over well known robust algorithms such as the Huber function, is in its speed, which when dealing with large quantities of data is paramount. It does so by paralizing the data in way that creates subproblems. The gradient descent, the most computationally heavy portion of learning, can then be applied on smaller but representative data that solves the general problem.

When applied to iid data, the MOM algorithm partitions the data into k equal sized groups or blocks then returns the median of the sample means of each of the k groups. However in this case, we're working with dependent data which presents its own issues.

The primary way in which we circumvented the issue of dependency in an AR(1) is by relying on the fact that the error terms are iid. This allows us to shuffle our data like we do in the iid case.

In general, we begin by taking our AR(1) of size n and calculate our epsilons which will be iid and gaussian, based on our setup. We will then divide the values into k blocks, which will be chosen in a way that optimizes our function, upon which we will take the mean of the epsilons within each block. We then take the median of the means of the blocks. The time series snippet which belongs to the errors which belong to the block, $B_{k_{med}}$ will be used to find our optimal $\hat{\theta}$ using gradient descent.

Our optimization function in which we aim to minimize our loss to find an optimal theta is:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\, med\{\frac{1}{|B_k|}\sum_{i \in B_k}(Y_t - \theta Y_{t-1})^2\} \tag{1}$$

Implementation

Shuffling data is important because it forces different combinations of data to be part of a chosen block. This allows us to better evaluate the performance of various block sizes in a way that doesn't depend on the combination of data that happens to be in a block at a given time.

```
data_shuffle <- function(x = x_ar1_out, k = blocks){
  #gen resids
  epsilon <- tail(x,-1) - theta_initial*head(x,-1)
  n <- length(x_ar1_out) / k

  sample_index <- sample(length(epsilon))
  epsilon_in <- epsilon[sample_index]
```

```
  data_head <- head(x_ar1_out, -1)[sample_index]
  data_head_block <- split(head(x_ar1_out, -1),ceiling(seq_along(epsilon_in)/n))

  data_tail <- tail(x_ar1_out, -1)[sample_index]
  data_tail_block <- split(tail(x_ar1_out, -1),ceiling(seq_along(epsilon_in)/n))

  #median of means
  #separates into blocks
  epsilon_block <- split(epsilon_in, ceiling(seq_along(epsilon_in)/n))
  epsilon_mean <- sapply(epsilon_block, function(x) mean(x), USE.NAMES = FALSE)
  #finds the index of the medianth block
  index <- match(epsilon_mean[order(epsilon_mean) == ceiling(k/2)][1], epsilon_mean)

  #grabbing appropriate block of epsilons / data you want to work with
  x_head_out <- unlist(data_head_block[index], use.names = FALSE)
  x_tail_out <- unlist(data_tail_block[index], use.names = FALSE)

  list(h = x_head_out, t = x_tail_out)
}
```

Below is the Median of Means algorithm described above. For the moment the results are not those that we expect nor are they reliable. My next step is to implement stochastic gradient descent. For the moment, depending on the hyperparameters and parameters chosen, the algorithm performs as well as the Huber function and Least Squares without the presence of outliers, and outperforms both in terms of accuracy in the presence of outliers. In both cases, the variance of MOM is large. This is one of its most important limitations.

```
block_by_thetas <- list()
pool <- choose_values(num_samples_train)
thetas <- seq(from = 0.1, to = 0.9, by = 0.1)
num_iterations <- 1000
number_outliers <-5
num_samples_train <- 1000
num_samples_test <- 500
theta_it_MOM_tails <-c()
lambda = 15

for(blocks in pool){
  for(theta_val in thetas){
    set.seed(1)
    x_ar1_out <- ar1_outliers(n_sample = num_samples_train,
                              n_outliers = number_outliers,
                              theta = theta_val,
                              method = "gaussian")

    data_block_test <- data_split(n_sample = num_samples_test,
                                  k = blocks, n_outliers = number_outliers,
                                  theta = theta_val, method = "gaussian")
    x_head_out_test <- data_block_test$h
    x_tail_out_test <- data_block_test$t

      for(i in 2:num_iterations){
        eta <- 1 / (1 + lambda * i)
        # print(eta)
```

```
        data_block <- data_shuffle(x_ar1_out, blocks)
        x_head_out <- data_block$h
        x_tail_out <- data_block$t
        #use your own gradient descent
        theta_it_MOM[i] <- theta_it_MOM[i-1] +
          eta * sum(x_head_out*(x_tail_out - theta_it_MOM[i-1] * x_head_out))

      }
      theta_it_MOM_tails <- tail(theta_it_MOM, 1)
      mse_MOM[(theta_val * 10)] <- (sum((x_tail_out_test -
                            tail(theta_it_MOM,1)*x_head_out_test)^2))/length(x_head_out_test)
      block_by_thetas <- rbind(block_by_thetas,
                               data.frame(theta_val, blocks, mse_MOM[(theta_val * 10)]))
    }
}
```