SparkRMA Documentation

Release 0.2.2

Michael T. Neylon

May 07, 2018

Contents:

CHAPTER 1

Getting Started with SparkRMA

SparkRMA: a scalable method for preprocessing oligonucleotide arrays

- Michael T. Neylon and Naveed Shaikh

1.1 Background

Robust Multi-array Average (RMA) is a method for preprocessing oligonulceotide arrays. Refer to the canonical publications on this topic:

- B M Bolstad et al. "A comparison of normalization methods for high density oligonucleotide array data based on variance and bias". In: Bioinformatics 19.2 (Jan. 2003), pp. 185–93.
- Rafael A Irizarry et al. "Exploration, normalization, and summaries of high density oligonucleotide array probe level data". In: Biostatistics 4.2 (Apr. 2003), pp. 249–64. doi: 10.1093/biostatistics/4.2.249.
- Rafael A Irizarry et al. "Summaries of Affymetrix GeneChip probe level data". In: Nucleic acids research 31.4 (2003), e15–e15.

1.2 Requirements

- Spark 2.0.0+

1.3 Installation

The scripts are intended to be submitted directly to the Spark driver using `spark-submit`, so no installation of SparkRMA is necessary. To run tests or build docs we have provided a setup.py script to install `spark_rma`.

1.4 AWS

A fast way to start using SparkRMA on large data sets is to use Amazon's Elastic MapReduce. Select an EMR version that meets the requirements listed above. There are dependencies that need to be installed and made available on each worker node. Currently that includes:

- pandas

Write a bootstrap script and store it in S3 to install pandas on each node:

```
sudo pip install pandas
```

CHAPTER 2

Overview

2.1 Annotation and Background Correction

Annotation and background correction are not done in Spark, because they are done at an individual array level and not dependent on other arrays. This means it is not a data-limiting step in RMA and makes it easy to parallelize. Annotation is the process of mapping probes to their perfect match (PM) targets. That is typically a transcript cluster or probeset.

R scripts are used for annotation and background correction, with the help of some Bioconductor packages. We also convert the output of this step into parquet for consumption in the following Spark steps.

2.2 Quantile Normalization

Quantile normalization removes array effects by normalizing each array against all others. This is a memory intensive step that requires all the arrays be evaluated at the same time. This is done using Spark window functions. The output of the values are log transformed (log2) at the start of the next step.

2.3 Summarization via Median Polish

Summarization takes the normalized, log-transformed probe values and summarizes them to their target to acquire an expression value for the target (a transcript cluster or probeset). This step groups all the same probes for a target from all samples before assigning the resulting expression values to each sample for that same target. This makes it a highly iterative, high memory step. We do this step using Spark UDF's to group the probes across samples and carry out the median polish using pandas and NumPy.

CHAPTER 3

Walkthrough an Example

HTA 2.0 are high-density microarrays that will be used for this example. This walk-through will also provide some detailed instructions using Amazon Web Services to step through all the parts of SparkRMA. These examples will use a small cluster of 4 workers of r4.8xlarge and 1 master node of r4.2xlarge. For all configuration options and arguments of each script, see the READMEs in GitHub or the command-line interface help.

3.1 Data

908 HTA 2.0 Samples from the Gene Expression Omnibus (GEO) from a Lilly Clinical Trial on Systemic Lupus Erythematosus (SLE).

3.2 CfnCluster

AWS CloudFormation Cluster is used to start a SunGrid Engine cluster for the annotation and background correction steps. Array jobs are a suitable method for completing this step. If you have a grid engine cluster available, the same directions could be used there.

Follow the instructions on setting up and using CfnCluster from the official docs.

3.3 EMR

AWS Elastic MapReduce is used for the quantile normalization and median polish steps. This provides a Hadoop/YARN cluster for deploying Spark. See the getting started page at Amazon.

Continue onto the next page to follow the tutorial

CHAPTER 4

Walkthrough: Annotation and Background Correction

4.1 Configure and Start CfnCluster

After installing CfnCluster according to the official docs listed above, setup your cluster config. For this example, use a 4 worker node cluster of r4.8xlarge and 1 master node of r4.2xlarge. Setup a CfnCluster config (typically located at ~/.cfncluster/config) with the following parameters (fill in your own values where there are asterisks):

```
[aws]
aws_region_name = ****
aws_access_key_id = *****
aws_secret_access_key = *****

[cluster default]
vpc_settings = public
key_name = *****
master_instance_type = r4.2xlarge
compute_instance_type = r4.8xlarge
initial_queue_size = 4
max_queue_size = 4
maintain_initial_size = true
ebs_settings = custom
master_root_volume_size = 50
compute_root_volume_size = 100

[vpc public]
master_subnet_id = *****
vpc_id = *****
vpc_security_group = *****

[global]
update_check = true
sanity_check = true
cluster_template = default
```

(continues on next page)

(continued from previous page)

```
[ebs custom]
volume_size = 2000
```

Tip: To save money, you can set the `initial_queue_size` and `max_queue_size` to 0 initially while you install software and download data, then change the values and update the cluster to start the workers.

- Start the cluster:
  - `cfncluster create sparkrma`
- SSH into the master. The Master IP or Public DNS should be output by CfnCluster when it's ready.
  - `ssh -i <PEM file> ec2-user@$PUBLIC_DNS` # SSH Key File should match the key name specified in config above

4.2 Setup Software on the Grid Engine Cluster

A few R/Bioconductor and Python packages need to be installed on the cluster.

This example shows how to use packrat and conda to make shared R and Python environments.

4.2.1 R

Install R and some dependencies:

`sudo yum -y install libcurl libcurl-devel openssl-devel R`

create packrat environment and install packages:

```
cd /shared/
mkdir renv
cd renv
R
> install.packages('devtools')
> devtools::install_github('rstudio/packrat')
> packrat::init()
> setRepositories()  # select 1 2 3
> install.packages('preprocessCore')
> install.packages("affyio")
> install.packages("docopt")
> install.packages('pd.hta.2.0')
```

4.2.2 Python

Install Miniconda:

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
sh Miniconda3-latest-Linux-x86_64.sh
```

During the setup:

- change install location to /shared/miniconda3/
- choose append to .bashrc

Then source it and install some packages that SparkRMA needs for converting the output to snappy-parquet format:

```
source ~/.bashrc
conda install fastparquet python-snappy
```

4.3 Download Data and SparkRMA

Download GSE88885:

```
cd /shared/
mkdir gse88885
cd gse88885
wget ftp://ftp.ncbi.nlm.nih.gov/geo/series/GSE88nnn/GSE88885/suppl/GSE88885_RAW.tar
mkdir raw
tar xvf GSE88885_RAW.tar -C raw/
```

Download SparkRMA:

```
cd /shared/
git clone https://github.com/michaeltneylon/spark_rma.git
```

Edit /shared/spark_rma/spark_rma/backgroundCorrect.R and /shared/helper/hta_annotation.R to make the R packrat environment available to it. At top of these files, on a line after the shebang, add:

```
source("/shared/renv/.Rprofile", chdir=TRUE)
```

4.4 Annotation, Background Correction, and Conversion to Parquet

First, generate the annotation file:

```
cd /shared/spark_rma/helper
./hta_annotation.R
```

Write a bash script to submit as an array job on the cluster, one job per sample. This is going to do annotation, with target specified as `-a` (annotation_type) with options `ps` or `tc`. In this example, `tc` is selected for transcript cluster annotation. Then the same script completes background correction and then converts the result to parquet.

Make our output directory before the job: `mkdir -p /shared/gse88885/background_corrected`

Write a shell script, let's call this gse88885.sh:

```
#! /bin/bash
#$ -t 1-908
#$ -cwd

input_dir=/shared/gse88885/raw
output_dir=/shared/gse88885/background_corrected
scratch=/scratch
bg_correct_home=/shared/spark_rma/spark_rma/
convert_home=/shared/spark_rma/helper
probe_list=/shared/spark_rma/helper/pm_probe_annotation.rda

# setup environment
export PATH="/shared/miniconda3/bin:$PATH"
```

(continues on next page)

(continued from previous page)

```
current_file=$(ls $input_dir/* | sed "${SGE_TASK_ID}q;d")
echo "processing $current_file"
output=${current_file##*/}
output=${output%.CEL.gz}
cd $bg_correct_home
rm -f $scratch/$output.background_corrected  # in case the file exists already from a␣
↪previous run.
./backgroundCorrect.R -p $probe_list -i $current_file -o $scratch/$output.background_
↪corrected -f text -a tc
sudo chmod 775 $scratch/$output.background_corrected
cd $convert_home
./convert_to_parquet.py $scratch/$output.background_corrected $output_dir/$output.
↪background_corrected.parquet
sudo rm $scratch/$output.background_corrected
```

This script can be anywhere since we defined absolute paths in it, so create it at $HOME/gse88885.sh

Then submit the job:

`qsub gse88885.sh`

4.5 Move to S3

Once the array job is finished, move the results into S3 using the aws cli. Using an existing bucket (create one if one does not already exist), copy the results to S3:

```
aws s3 cp \
  --recursive \
  /shared/gse88885/background_corrected/ \
  s3://<bucket name>/gse88885/background_corrected.parquet
```

The cluster may now be deleted: `cfncluster delete sparkrma`

CHAPTER 5

Walkthrough: Quantile Normalization and Summarization via Median Polish

5.1 Start EMR

An EMR cluster can be created in the console or through the aws cli.

First, create a bootstrap script in S3 for installing some dependencies:

Create a script at s3://<bucket_name>/bootstrap/install_dependencies.sh as:

```
#! /bin/bash

sudo pip install pandas
```

5.1.1 Console Setup

On the EMR console page, select 'Create Cluster', then select 'Go to advanced options'.

- Step 1: Software Configuration
  - Select EMR release 'emr-5.12.1' or up
  - For software select at least Hadoop and Spark
- Step 2: Hardware Configuration
  - Select 'uniform instance groups'
  - Adjust Root device EBS volume size to max '100 GiB'
  - For Master node, select r4.2xlarge with 200 GiB EBS Storage
  - For Core nodes, select r4.8xlarge with 500 GiB EBS Storage and adjust instance count to 4
- Step 3: General Cluster Settings
  - Cluster name: SparkRMA - GSE88885

  - Tags:
    - Key: Name; Value: SparkRMA - GSE88885
  - Additional Options
    - Bootstrap Actions
      - Add a bootstrap action:
      - select custom action
      - select configure and add
      - name it 'Install Pandas'
      - add the s3 path to the bootstrap script created earlier:
      - s3://<bucket_name>/bootstrap/install_dependencies.sh
- Step 4: Security Options
  - Select your EC2 Key Pair Name
  - select the default permissions
  - EC2 Security Groups
    - for Master, and optionally Core & Task, add an additional security group that has port 22 open for SSH
- Create Cluster

5.2 Copy Data into HDFS

Once the cluster is ready in 'waiting' status, ssh into the master node. The console will output the master's Public DNS.

```
ssh -i <ssh key file> hadoop@<master_public_DNS>
```

Use `distcp` to copy the annotated, background corrected data in parquet format from S3 to HDFS:

```
hadoop distcp \
  -Dmapreduce.map.memory.mb=5000 \
  s3://<bucket_name>/gse88885/background_corrected.parquet \
  background_corrected.parquet
```

5.3 Download SparkRMA

```
git clone https://github.com/michaeltneylon/spark_rma.git
```

5.4 Quantile Normalization

```
spark-submit \
  --driver-memory=55G \
  --executor-memory=10G \
  --conf spark.yarn.executor.memoryOverhead=1500M \
```

(continues on next page)

(continued from previous page)

```
  --conf spark.dynamicAllocation.maxExecutors=128 \
  --conf spark.sql.shuffle.partitions=908 \
  spark_rma/spark_rma/quantile_normalization.py \
  -i background_corrected.parquet \
  -o quantile_normalized.parquet
```

You can view the progress of the job through the Spark UI. Get the link through the Yarn ResourceManager UI. See this EMR page on web interface URIs

5.5 Median Polish

Using the same EMR cluster from the previous step of quantile normalization, submit the median polish job:

```
spark-submit \
  --driver-memory=55G \
  --executor-memory=8G \
  --conf spark.yarn.executor.memoryOverhead=1500M \
  --conf spark.dynamicAllocation.maxExecutors=128 \
  spark_rma/spark_rma/median_polish.py \
  -i quantile_normalized.parquet \
  -o tc_expression.parquet \
  -ns 908 -rn 30000
```

5.6 Move Results to S3

```
hadoop distcp \
  -Dmapreduce.map.memory.mb=5000 \
  tc_expression.parquet \
  s3://<bucket_name>/gse88885/tc_expression.parquet
```

The EMR cluster can now be terminated.

CHAPTER 6

spark_rma

6.1 spark_rma package

6.1.1 Submodules

6.1.2 spark_rma.median_polish module

Carry out median polish on grouping of probes, to return a value for each sample in that grouping.

class spark_rma.median_polish.**ProbesetSummary**(*spark*, *input_data*, *num_samples*, *repartition_number*)

Bases: *spark_rma.median_polish.Summary*

Summarize probes with probeset region grouping

class spark_rma.median_polish.**Summary**(*spark*, *input_data*, *num_samples*, *repartition_number*, ***kwargs*)

Bases: object

Summarize gene expression values via median polish.

summarize()

Summarize results across samples with median polish within defined groups.

udaf(*data*)

Apply median polish to groupBy keys and return value for each sample within that grouping.

This is a hacked/workaround user-defined aggregate function (UDAF) that passes the grouped data to python to do median polish and return the result back to the dataframe.

Returns spark dataframe

class spark_rma.median_polish.**TranscriptSummary**(*spark*, *input_data*, *num_samples*, *repartition_number*)

Bases: *spark_rma.median_polish.Summary*

Summarize probes with transcript cluster groupings

spark_rma.median_polish.**command_line**()
Collect and validate command line arguments.

spark_rma.median_polish.**infer_grouping_and_summarize**(*spark*, *input_file*, *output_file*, *num_samples*, *repartition_number*)
Read the input file to infer grouping type and select appropriate summarization class.

spark_rma.median_polish.**main**()
Collect command-line arguments and start spark session when using spark-submit.

spark_rma.median_polish.**probe_summarization**(*grouped_values*)
Summarization step to be pickled by Spark as a UDF. Receives a groupings data in a list, unpacks it, performs median polish, calculates the expression values from the median polish matrix results, packs it back up, and return it to a new spark Dataframe.

Parameters **grouped_values** – a list of strings because spark concatenated all the values into one string for each sample. Each item is a sample,probe, value format and all the rows in the input belong to a grouping key that spark handled (transcript_cluster or probeset)

Returns a list of lists where each item is length two with (sample, value)

6.1.3 spark_rma.quantile_normalization module

Quantile normalize background corrected array samples.

spark_rma.quantile_normalization.**command_line**()
Collect and validate command line arguments.

spark_rma.quantile_normalization.**infer_target_level**(*data_frame*)
Read the input file to infer target type and select appropriate summarization class.

spark_rma.quantile_normalization.**main**()
Gather command-line arguments and create spark session if executed with spark-submit

spark_rma.quantile_normalization.**normalize**(*spark*, *input_path*, *output*)
Read parquet file, normalize, and write results.

Parameters

- **spark** – spark session object
- **input_path** – path to input parquet file with background corrected data
- **output** – path to write results

spark_rma.quantile_normalization.**quantile_normalize**(*data_frame*, *target*)
Quantile normalize the data using spark window spec. This allows the ranking, average, and reassigning values with a join.

Parameters

- **data_frame** – spark data frame
- **target** – summary target defined by annotation. This is the level to which we will summarize, either probeset or transcript_cluster.

Returns quantile normalized dataframe with sample, probe, target, and normalized value.

6.1.4 Module contents

6.2 helper package

6.2.1 Submodules

6.2.2 helper.convert_to_parquet module

Converts files into snappy-parquet without using Spark and JVM. This is so they are compressed before putting into HDFS and/or before Spark reads them to accelerate all these tasks.

class helper.convert_to_parquet.**Conversion**(*input_name*, *output_name*, *delimiter*)

Convert file to parquet.

check()

Check if the output file has been written before already. During recovery or re-run, don't waste time by re-writing the same files.

Returns bool if path exists

execute()

Check the file's existence and convert into parquet if the output does not already exist. Public method to call on the class.

read()

Read the files into Pandas dataframe, required by fastparquet. Reads based on delimiter, faster to explicitly switch between these types than to detect since it uses the c engine instead of python to read.

Returns pandas.dataframe()

write(*dataframe*)

Write a pandas dataframe into parquet format using fastparquet and snappy compression.

Parameters **dataframe** – a pandas dataframe as input

helper.convert_to_parquet.**call_conversion**(*in_name*, *out_name*, *mode*)

Create object and call method to execute conversion. This is required so that multiprocessing can pickle a function, it cannot pickle a class.

Parameters

- **in_name** – input file name in flat file, tsv or csv
- **out_name** – output file name in parquet
- **mode** – delimiter, comma or tab.

helper.convert_to_parquet.**command_line**()

Collect and validate command line arguments.

helper.convert_to_parquet.**makedir_if_not_exist**(*directory*)

Create a directory if it does not already exist.

Parameters **directory** – (str) Directory name.

helper.convert_to_parquet.**parallelizer**(*input_directory*, *output_directory*, *mode*)

Parallelizes this program to convert a whole directory of files.

Parameters

- **input_directory** – input directory of flat files
- **output_directory** – name of directory to write to

- **mode** – delimiter in files, currently must be the same for whole directory

6.2.3 Module contents

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

s

Index