VariantS Documentation

Release 0.2.0

Piotr Szul

Nov 08, 2019

Contents

variant-spark is a scalable toolkit for genome-wide association studies optimized for GWAS like datasets.

Machine learning methods and, in particular, random forests (RFs) are a promising alternative to standard single SNP analysis in genome-wide association studies (GWAS). RFs provide variable importance measures to rank SNPs according to their predictive power. Although there are number of existing random forest implementations available, some even parallel or distributed such as: Random Jungle, ranger or SparkML, most of them are not optimized to deal with GWAS datasets, which usually come with thousands of samples and millions of variables.

variant-spark currently provides the basic functionality of building random forest model and estimating variable importance with mean decrease gini method and can operate on VCF and CSV files. Future extensions will include support of other importance measures, variable selection methods and data formats.

variant-spark utilizes a novel approach of building random forest from data in transposed representation, which allows it to efficiently deal with even extremely wide GWAS datasets. Moreover, since the most common genomics variant calls VCF and uses the transposed representation, variant-spark can work directly with the VCF data, without the costly pre-processing required by other tools.

variant-spark is built on top of Apache Spark – a modern distributed framework for big data processing, which gives variant-spark the ability to to scale horizontally on both bespoke cluster and public clouds.

The potential users include:

- Medical researchers seeking to perform GWAS-like analysis on large cohort data of genome wide sequencing data or imputed SNP array data.
- Medical researchers or clinicians seeking to perform clustering on genomic profiles to stratify large-cohort genomic data
- General researchers with classification or clustering needs of datasets with millions of features.

CHAPTER 1

Contents:

1.1 Getting Started

VariantSpark is currently supported only on Unix-like system (Linux, MacOS).

You'll need:

- The Java 8 JDK.
- Apache Spark 2.2.1. **VariantSpark** is compatible with Spark 2.1+
- Python 2.7 and Jupyter Notebooks. We recommend the free Anaconda distribution.

It's easy to run locally on one machine - you need to have Apache Spark installed correctly with:

- `spark-submit` and `pyspark` on you system `PATH`

1.1.1 Installing from a distribution

Get **VariantSpark** distribution from the downloads page of project web site. Un-tar the distribution after you download it (you may need to change the name of the file to match the current version).

```
tar -xzf variant-spark_2.11-0.2.0.tar.gz
```

Next, edit and copy the below bash commands to set up the **VariantSpark** environment variables. You may want to add these to the appropriate dot-file (we recommend `~/.profile`) so that you don't need to rerun these commands in each new session.

Here, fill in the path to the un-tared **VariantSpark** distribution.

```
export VARSPARK_HOME=???
export PATH=$PATH:$VARSPARK_HOME/bin
```

Now you should be all setup to run **VariantSpark** command line tool.

```
variant-spark -h
```

The *-h* option displays the help on available commands. To find out more about the command line tool please visit *Command Line Reference*.

VariantSpark comes with several sample programs and datasets. Command-line and Python Jupyter examples are `examples` directory. There is a few small data sets in the `data` directory suitable for running on a single machine.

1.1.2 Installing from PyPI

It's recommended that Python users install **VariantSpark** from PyPI with:

```
pip install variant-spark
```

This assumes that a compatible version of Apache Spark is already installed in your Python environment. If not, you can install it from the distribution using the information from the beginning of this section, or with:

```
pip install variant-spark[spark]
```

The code and data samples are installed into the `<PYTHON_PREFIX>/share/variant-spark` directory, where `<PYTHON_PREFIX>` is he value of Python's `sys.prefix`.

You can find what your `<PYTHON_PREFIX>` is on your system by typing:

```
python -c $'import sys\nprint(sys.prefix)'
```

It's recommend that you make a copy the examples in your home/working directory before using them.

1.1.3 Running examples

The rest of this section assumes that you are in the `examples` directory.

One of the main applications of **VariantSpark** is discovery of genomic variants correlated with a response variable (e.g. case vs control) using random forest gini importance.

The `chr22_1000.vcf` is a very small sample of the chromosome 22 VCF file from the 1000 Genomes Project.

`chr22-labels.csv` is a CSV file with sample response variables (labels). In fact the labels directly represent the number of alternative alleles for each sample at a specific genomic position. E.g.: column `22_16050408` has labels derived from variants in chromosome 22 position 16050408. We would expect then that position *22:16050408* in the VCF file is strongly correlated with the label `22_16050408`.

Running importance analysis with command line tool

We can run importance analysis on these data with the following command:

```
variant-spark importance -if ../data/chr22_1000.vcf -ff ../data/chr22-labels.csv -fc␣
↪22_16050408 -v -rn 500 -rbs 20 -ro -sr 13
```

alternatively you can just the sample script:

```
./local_run-importance-ch22.sh
```

This will build a random forest with 500 trees and report OOB (out of bag) error as well as the top 20 important variables. The final output should be similar to this:

```
Random forest oob accuracy: 0.015567765567765568, took: 28.249 s
variable,importance
22_16050408,8.634503639847714E-4
22_16051107,7.843083422549387E-4
...
```

As expected variable `22_16050408` representing the variant at position *22:16050408* comes out as the most important.

To find out about the options for the *importance* command type `variant-spark importance -h` or visit *Command Line Reference*.

Running importance analysis with PythonAPI

The same analysis can be also performed using **VariantSpark** PythonAP with a Jupyter notebook.

You can start the *VariantSpark* enabled Jupyter notebook server with:

```
jvariant-spark
```

Open an run the `run_importance_chr22.ipynb` for the PythonAPI example. You can check the expected results here.

To find out about more about the Python API visit *Python API*.

1.1.4 Where to Go from Here

- If you'd like to build VariantSpark from source, visit *Development*.

1.2 Overview

TBP: Info on random forest, importance and how it's applicable in GWAS. Comand line tool and various apis.

1.2.1 Importance analysis

`TBP`

`Some content that may go here:`

Prepare VCF file for VariantSpark importance analysis:

VariantSpark encodes 0/0, 0/X and X/Y genotypes into 0, 1 and 2 respectively where X and Y refer to any of alternate allele in the ALT field (>=1). This could be a problem when dealing with multiallelic variants. You may decide to breakdown multi-allelic sites into multiple bi-allelic sites prior to the importance analysis (both BCFtools and Hail provide this functionality). This allows you to correctly consider the effect of each allele (on the same site) separately. In case you need to keep all allele in the same variable and at the same time distinguish between different allele, you can manually transform the VCF file in a CSV file and use your own encoding. Note that having more distinct values for a variable will slow down the RandomForest training process. Also, note that VariantSpark does not consider phasing in the VCF file. Thus both 0|1 and 1|0 are encoded to 1 assumed to be the same.

Another consideration is about joining VCF file together (i.e. join the case and control VCF file). Hail join function ignores multi-allelic sites with different allele lists. BCFtools join multi-allelic sites but cause errors. We strongly

recommend to breakdown multi-alleic site before joining VCF file and if needed merge them to the same site after joining (both BCFtools and Hail provide this functionality).

Some notes for using VariantSpark importance analysis:

The value of mtry significantly affects the RandomForest training process. The default value of mtry is the square root of the number of variables. Although it is a good value, in general, it does not appropriate for all sorts of data. You may consider varying the value of mtry and re-run the analysis to find the optimal value that suits your dataset. Note that with lower mtry processing each node of a tree would be faster but trees will get deeper. Thus, the relation between mtry and execution time is not linear.

There are some papers about how to estimate number of trees.

The most applicable approach is to run with 10000 and 20000 trees (with default mtry) if there were significant changes in the top 100 or top 1000 important variable the model is not stable and they should try with 40000 and so on.

I recommend to run with mtry = 0.1 * num_variants (with 10000 trees) as this may change the top hits.

I also recommend to run with low mtry (i.e. 100) and as many trees as possible (500000) as this may change the top hits too (with low mtry tree building is much faster)

1.3 Installation

1.3.1 Running locally

1.3.2 Running on YARN cluster

1.3.3 Running on Amazon EMR

`TODO:` Copy content from: https://github.com/aehrc/VariantSpark/tree/master/cloud/aws-emr

1.3.4 Running on Databricks

1.4 Tutorials

1.4.1 ??? Importance Analysis with command line

TBP:

1.4.2 Introduction to Python API

This notebook demostrates the use of Variant-Spark Python API for importance analysis.

Introduction to Python API

[1]:
```
from varspark import VariantsContext
```

[2]:
```
vc = VariantsContext(spark)
```

```
Running on Apache Spark version 2.2.1
SparkUI available at http://140.253.176.47:4040
Welcome to
 _    __           _              __  ______                    __
| |  / /___ _____(_)___ _____  / /_/ ___/____  ____ _____/ /__
| | / / __ `/ ___/ / __ `/ __ \/ __/\__ \/ __ \/ __ `/ ___/ //_/
| |/ / /_/ / /  / / /_/ / / / / /_ ___/ / /_/ / /_/ / /  / ,<
|___/\__,_/_/  /_/\__,_/_/ /_/\__//____/ .___/\__,_/_/  /_/|_|
                                      /_/
```

[3]:
```
fs = vc.import_vcf('../data/chr22_1000.vcf')
```

[4]:
```
ls = vc.load_label('../data/chr22-labels.csv', '22_16050408')
```

Running importance analysis

[5]:
```
ia = fs.importance_analysis(ls, mtry_fraction=0.1, seed = 13)
```

[6]:
```
ia.important_variables()
```

[6]:
```
[(u'22_16050408', 0.0017817043235329407),
 (u'22_16050678', 0.0010904805815147934),
 (u'22_16051480', 0.0010668521995151252),
 (u'22_16051107', 0.0008082666531272816),
 (u'22_16052838', 0.0006518182467402832),
 (u'22_16053197', 0.0005765782334748871),
 (u'22_16052656', 0.0004967689362074311),
 (u'22_16051882', 0.0004909085098234914),
 (u'22_16053435', 0.0004820979788328268),
 (u'22_16053727', 0.0004199166942675162)]
```

1.4.3 ??? GWS with python API

TBP: possibly extracted from the databricks notebook with references to it. - actually it's possible to include actual ipython notebooks here, so the some of the tutorials can be in this for (but I not sure about the datbrick notebooks)

1.5 Command Line Reference

variant-spark requires an existing spark 2.1+ installation (either a local one or a cluster one).

To run variant-spark use:

```
./variant-spark [(--spark|--local) <spark-options>* --] [<command>] <command-options>*
```

In order to obtain the list of the available commands use:

```
./variant-spark -h
```

In order to obtain help for a specific command (for example *importance*) use:

```
./variant-spark importance -h
```

You can use `--spark` marker before the command to pass *spark-submit* options to variant-spark. The list of spark options needs to be terminated with `--`, e.g:

./variant-spark –spark –master yarn-client –num-executors 32 – importance ....

Please, note that `--spark` needs to be the first argument of **variant-spark**

You can also run variant-spark in the `--local` mode. In this mode variant-spark will ignore any Hadoop or Spark configuration files and run in the local mode for both Hadoop and Spark. In particular in this mode all file paths are interpreted as local file system paths. Also any parameters passed after *–local* and before – are ignored. For example:

```
./variant-spark --local -- importance  -if data/chr22_1000.vcf -ff data/chr22-labels.
↪csv -fc 22_16051249 -v -rn 500 -rbs 20 -ro
```

Note:

The difference between running in `--local` mode and in `--spark` with `local` master is that in the latter case Spark uses the hadoop filesystem configuration and the input files need to be copied to this filesystem (e.g. HDFS) Also the output will be written to the location determined by the hadoop filesystem settings. In particular paths without schema e.g. 'output.csv' will be resolved with the hadoop default filesystem (usually HDFS) To change this behavior you can set the default filesystem in the command line using *spark.hadoop.fs.default.name* option. For example to use local filesystem as the default use:

```
variant-spark --spark ... --conf "spark.hadoop.fs.default.name=file:///" ... --␣
↪importance  ... -of output.csv
```

You can also use the full URI with the schema to address any filesystem for both input and output files e.g.:

```
variant-spark --spark ... --conf "spark.hadoop.fs.default.name=file:///" ... --␣
↪importance  -if hdfs:///user/data/input.csv ... -of output.csv
```

TBP: Reference for running command line: - add prerequisites, Spark, SPARK_HOME, Spark Bin on path - de-empasise the local mode - should also include options and file formats used

1.6 Python API

This is the API documentation for `VariantSpark`, and provides detailed information on the Python programming interface.

The code below illustrates the basic use of variant-spark:

```
from varspark import VariantsContext
from pyspark.sql import SparkSession

spark = SparkSession.builder\
    .appName("HipsterIndex") \
    .getOrCreate()

vs = VariantsContext(spark)
features = vs.import_vcf(VCF_FILE)
labels = vs.load_label(LABEL_FILE, LABEL_NAME)
model  = features.importance_analysis(labels, mtry_fraction = 0.1, seed = 13, n_trees␣
↪= 200)
print("Oob = %s" % model.oob_error())
```

Contents:

1.6.1 package varspark

This module includes the core variant-spark API.

1.6.2 module varspark.core

This module includes the core variant-spark API.

class varspark.core.**FeatureSource**(*_jvm*, *_vs_api*, *_jsql*, *sql*, *_jfs*)

importance_analysis(***kwargs*)
Builds random forest classifier.

Parameters

- **label_source** – The ingested label source
- **n_trees** (*int*) – The number of trees to build in the forest.
- **mtry_fraction** (*float*) – The fraction of variables to try at each split.
- **oob** (*bool*) – Should OOB error be calculated.
- **seed** (*long*) – Random seed to use.
- **batch_size** (*int*) – The number of trees to build in one batch.
- **var_ordinal_levels** (*int*) –

Returns Importance analysis model.

Return type *ImportanceAnalysis*

class varspark.core.**ImportanceAnalysis**(*_jia*, *sql*)
Model for random forest based importance analysis

important_variables(***kwargs*)
Gets the top limit important variables as a list of tuples (name, importance) where: - name: string - variable name - importance: double - gini importance

oob_error()
OOB (Out of Bag) error estimate for the model

Return type float

variable_importance()
Returns a DataFrame with the gini importance of variables.

The DataFrame has two columns: - variable: string - variable name - importance: double - gini importance

varspark.core.**VariantsContext**
alias of *varspark.core.VarsparkContext*

class varspark.core.**VarsparkContext**(*ss*, *silent=False*)
The main entry point for VariantSpark functionality.

import_vcf(***kwargs*)
Import features from a VCF file.

load_label(***kwargs*)
Loads the label source file

Parameters

- **label_file_path** – The file path for the label source file
- **col_name** – the name of the column containing labels

classmethod spark_conf(*conf=<pyspark.conf.SparkConf object>*)
Adds the necessary option to the spark configuration. Note: In client mode these need to be setup up using –jars or –driver-class-path

stop()
Shut down the VariantsContext.

1.6.3 package varspark.hail

This package contains variant spark integration with Hail.

```
from hail import *
import varspark.hail
hc = HailContext(sc)
vds = hc.import_vcf(...)
...
via = vds.importance_analysis("sa.pheno.label", n_trees = 1000)
```

module varspark.hail.extend

Created on 7 Nov 2017

@author: szu004

class varspark.hail.extend.**VariantsDatasetFunctions**(**args*, ***kwargs*)
Extension to hail.VariantDataset with variant-spark related functions

importance_analysis(***kwargs*)
Builds random forest classifier for the response variable defined with y_expr.

Parameters

- **y_expr** (*str*) – Response expression. Must evaluate to Boolean or numeric with all values 0 or 1.
- **n_trees** (*int*) – The number of trees to build in the forest.
- **mtry_fraction** (*float*) – The fraction of variables to try at each split.
- **oob** (*bool*) – Should OOB error be calculated.
- **seed** (*long*) – Random seed to use.
- **batch_size** (*int*) – The number of trees to build in one batch.

Returns Importance analysis model.

Return type ImportanceAnalysis

pairwise_operation(***kwargs*)
Computes a pairwise operation on encoded genotypes. Currently implemented operations include:

- *manhattan* : the Manhattan distance

- *euclidean* : the Euclidean distance
- *sharedAltAlleleCount*: count of shared alternative alleles
- *anySharedAltAlleleCount*: count of variants that share at least one alternative allele

Parameters **operation_name** – name of the operaiton. One of *manhattan*, *euclidean*, *sharedAltAlleleCount*, *anySharedAltAlleleCount*

Returns A symmetric *no_of_samples x no_of_samples* matrix with the result of the pairwise computation.

Return type hail.KinshipMatrix

module varspark.hail.rf

Created on 10 Nov 2017

@author: szu004

class varspark.hail.rf.**ImportanceAnalysis**(*hc*, *_jia*)

Model for random forest based importance analysis

important_variants(***kwargs*)

Gets the top n most important loci.

Parameters **n_limit** (*int*) – the limit of the number of loci to return

Returns A KeyTable with the variant in the first column and importance in the second.

Return type hail.KeyTable

oob_error

OOB (Out of Bag) error estimate for the model

Return type float

1.6.4 module varspark.utils

This module includes various utility functions e.g. for converting objects between Hail, Spark, numpy and pandas.

Created on 6 Dec 2017

@author: szu004

varspark.utils.**array_to_dataframe**(*ndarray*, *labels=None*)

Converts a square numpy array to a pandas dataframe with index and column names from labels (if provided)

Parameters

- **ndarray** – a square numpy array to convert
- **labels** – labels to use for the index and for the column names

Returns a pandas dataframe

varspark.utils.**array_to_dataframe_coord**(*ndarray*, *labels=None*, *triangular=True*, *include_diagonal=True*, *row_name='row'*, *col_name='col'*, *value_name='value'*)

Converts a square numpy array to a pandas dataframe in coordinate format that is *[row, column, value]*. Optionally only includes the lower triangular matrix with or without diagonal (to get only unique coordinates)

Parameters

- **labels** – labels to use for row and columns coordinates
- **triangular** – only include the lower triangular matrix
- **include_diagonal** – if the main diagonal should be included
- **row_name** – the name to use for row column (first coordinate)
- **col_name** – the name to use for col column (second coordinate)
- **value_name** – the name to use for the value column

Returns dataframe with the values from the kinship matrix in the coordinate form

varspark.utils.**dist_mat_to_array**(*dist_mat*)
Converts a (small) distributed matrix to dense numpy narray

Parameters **dist_mat** – a pyspark.mllib.linalg distributed matrix

Returns a local numpy array with the matrix data

varspark.utils.**kinship_mat_to_dataframe**(*km*)
Converts a hail KinshipMatrix to a pandas dataframe. Index and column names are obtained from *sample_list* of the matrix.

Parameters **km** – kinship matrix to convert

Returns dataframe with the values from the kinship matrix

varspark.utils.**kinship_mat_to_dataframe_coord**(*km*, ***kwargs*)
Converts a hail KinshipMatrix to a pandas dataframe. Coordinate values are obtained from *sample_list* of the matrix.

Parameters

- **km** – kinship matrix to convert
- **kwargs** – other conversion parameters as in [[array_to_dataframe_coord]]

Returns dataframe with the values from the kinship matrix in the coordinate form

1.7 Development

variant-spark requires:

- java jdk 1.8+
- maven 3+

In order to build the binaries use:

```
mvn clean install
```

For python **variant-spark** requires python 2.7 with pip.

The other packages required for development are listed in dev/dev-requirements.txt and can be installed with:

```
pip install -r dev/dev-requirements.txt
```

or with:

```
./dev/py-setup.sh
```

The complete built including all check can be run with:

```
./dev/build.sh
```

TODO:

- how to make a distribution or build
- how to contirbute see: https://github.com/aehrc/VariantSpark/blob/master/CONTRIBUTING.md
- copy part of the contents from: https://github.com/aehrc/VariantSpark/blob/master/python/README.md

1.8 Indices and tables

- genindex
- modindex
- search

Python Module Index

V

Index