glow Documentation

Glow Project

Nov 02, 2021

CONTENTS

Glow is an open-source toolkit for working with genomic data at biobank-scale and beyond. The toolkit is natively built on Apache Spark, the leading unified engine for big data processing and machine learning, enabling genomics workflows to scale to population levels.

CHAPTER

ONE

INTRODUCTION TO GLOW

Genomics data has been doubling every seven months globally. It has reached a scale where genomics has become a big data problem. However, most tools for working with genomics data run on single nodes and will not scale. Furthermore, it has become challenging for scientists to manage storage, analytics and sharing of public data.

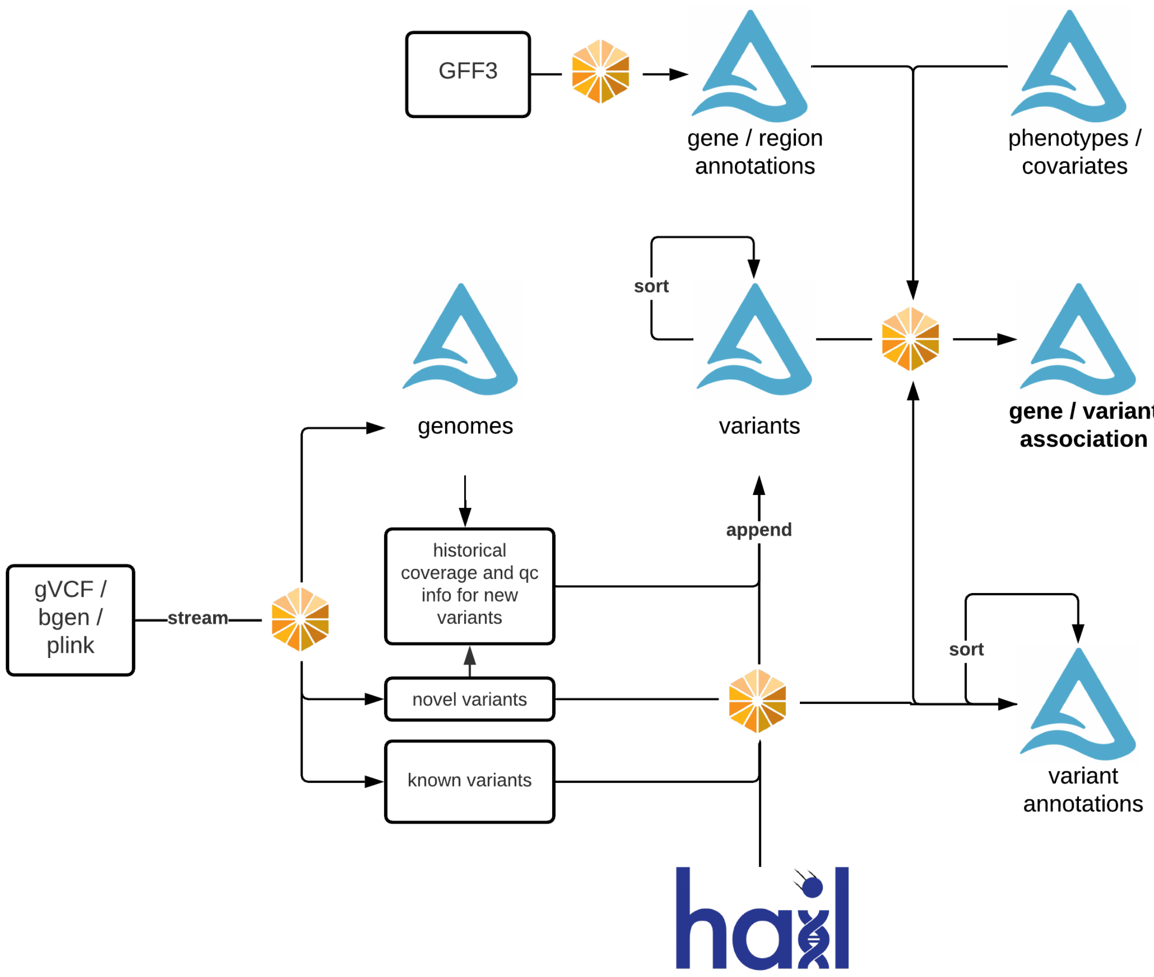
Glow solves these problems by bridging bioinformatics and the big data ecosystem. It enables bioinformaticians and computational biologists to leverage best practices used by data engineers and data scientists across industry.

Glow is built on Apache Spark and Delta Lake, enabling distributed computation on and distributed storage of genotype data. The library is backwards compatible with genomics file formats and bioinformatics tools developed in academia, enabling users to easily share data with collaborators.

When combined with Delta Lake, Glow solves the “n+1” problem in genomics, allowing continuous integration of and analytics on whole genomes without data freezes.

Glow is used to:

- Ingest genotype data into a data lake that acts as a single source of truth.
- Perform joint-genotyping of genotype data on top of delta-lake.
- Run quality control, statistical analysis, and association studies on population-scale datasets.
- Build reproducible, production-grade genomics data pipelines that will scale to tens of trillions of records.



Glow features:

- Genomic datasources: To read datasets in common file formats such as VCF, BGEN, and Plink into Spark DataFrames.
- Genomic functions: Common operations such as computing quality control statistics, running regression tests, and performing simple transformations are provided as Spark functions that can be called from Python, SQL, Scala, or R.
- Data preparation building blocks: Glow includes transformations such as variant normalization and lift over to help produce analysis ready datasets.
- Integration with existing tools: With Spark, you can write user-defined functions (UDFs) in Python, R, SQL, or Scala. Glow also makes it easy to run DataFrames through command line tools.
- Integration with other data types: Genomic data can generate additional insights when joined with data sets such as electronic health records, real world evidence, and medical images. Since Glow returns native Spark SQL DataFrames, its simple to join multiple data sets together.
- GloWGR, a distributed version of the regenie method, rewritten from the ground up in Python.

CHAPTER

TWO

GETTING STARTED

2.1 Running Locally

Glow requires Apache Spark 3.1.2.

Python

Scala

If you don't have a local Apache Spark installation, you can install it from PyPI:

```
pip install pyspark==3.1.2
```

or download a specific distribution.

Install the Python frontend from pip:

```
pip install glow.py
```

and then start the Spark shell with the Glow maven package:

```
./bin/pyspark --packages io.projectglow:glow-spark3_2.12:1.1.0 --conf spark.hadoop.io.
↪compression.codecs=io.projectglow.sql.util.BGZFCodec
```

To start a Jupyter notebook instead of a shell:

```
PYSPARK_DRIVER_PYTHON=jupyter PYSPARK_DRIVER_PYTHON_OPTS=notebook ./bin/pyspark --
↪packages io.projectglow:glow-spark3_2.12:1.1.0 --conf spark.hadoop.io.compression.
↪codecs=io.projectglow.sql.util.BGZFCodec
```

And now your notebook is glowing! To access the Glow functions, you need to register them with the Spark session.

```
import glow
spark = glow.register(spark)
df = spark.read.format('vcf').load(path)
```

If you don't have a local Apache Spark installation, download a specific distribution.

Start the Spark shell with the Glow maven package:

```
./bin/spark-shell --packages io.projectglow:glow-spark3_2.12:1.1.0 --conf spark.hadoop.
↪io.compression.codecs=io.projectglow.sql.util.BGZFCodec
```

To access the Glow functions, you need to register them with the Spark session.

```
import io.projectglow.Glow
val sess = Glow.register(spark)
val df = sess.read.format("vcf").load(path)
```

2.2 Notebooks embedded in the docs

To demonstrate use cases of Glow, documentation pages are accompanied by embedded notebooks. Most code in these notebooks can be run on Spark and Glow alone, but functions such as `display()` or `dbutils()` are only available on Databricks. See *Databricks notebooks* for more info.

Also note that the path to datasets used as example in these notebooks is usually a folder in `/databricks-datasets/genomics/` and should be replaced with the appropriate path based on your own folder structure.

2.3 Getting started on Databricks

The Databricks documentation shows how to get started with Glow on,

- **Amazon Web Services** (AWS - docs)
- **Microsoft Azure** (docs)
- **Google Cloud Platform** (GCP - docs)

2.4 Getting started on other cloud services

Please submit a pull request to add a guide for other cloud services.

CHAPTER

THREE

VARIANT DATA MANIPULATION

Glow offers functionalities to extract, transform and load (ETL) genomic variant data into Spark DataFrames, enabling manipulation, filtering, quality control and transformation between file formats.

3.1 Data Simulation

The data simulation notebooks below generate genotypes, phenotypes and covariates at a user-defined scale. This dataset can be used for integration and scale-testing.

3.1.1 Simulate Genotypes

This data simulation notebook downloads chromosomes **21** and **22** from the 1000 Genomes Project, and returns a Delta Lake table with a simulated set of genotypes for **n_samples** and **n_variants**, maintaining hardy-weinberg equilibrium and allele frequency for each variant.

Notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/simulate_delta_pvcf.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/simulate_delta_pvcf.html" id='-7903629770099432220' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.1.2 Simulate Covariates & Phenotypes

This data simulation notebooks uses Pandas to simulate quantitative and binary phenotypes and covariates. Please ensure **n_samples** is the same as the genotype simulation notebook above.

Notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/simulate_covariates_phenotypes_offset.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/simulate_covariates_phenotypes_offset.html" id='9171433679780647269' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.2 Read and Write VCF, Plink, and BGEN with Spark

Glow makes it possible to read and write variant data at scale using Spark SQL.

Tip: This topic uses the terms "variant" or "variant data" to refer to single nucleotide variants and short indels.

3.2.1 VCF

You can use Spark to read VCF files just like any other file format that Spark supports through the DataFrame API using Python, R, Scala, or SQL.

```
df = spark.read.format("vcf").load(path)
assert_rows_equal(df.select("contigName", "start").head(), Row(contigName='17', start=504217))
```

The returned DataFrame has a schema that mirrors a single row of a VCF. Information that applies to an entire variant (SNV or indel), such as the contig name, start and end positions, and INFO attributes, is contained in columns of the DataFrame. The genotypes, which correspond to the GT FORMAT fields in a VCF, are contained in an array with one entry per sample. Each entry is a struct with fields that are described in the VCF header.

The path that you provide can be the location of a single file, a directory that contains VCF files, or a Hadoop glob pattern that identifies a group of files. Sample IDs are not included by default. See the parameters table below for instructions on how to include them.

You can control the behavior of the VCF reader with a few parameters. All parameters are case insensitive.

Parameter	Type	Default	Description
`includeSampleIds`	boolean	`true`	If true, each genotype includes the name of the sample ID it belongs to. Sample names increase the size of each row, both in memory and on storage.
`flattenInfoFields`	boolean	`true`	If true, each info field in the input VCF will be converted into a column in the output DataFrame with each column typed as specified in the VCF header. If false, all info fields will be contained in a single column with a string -> string map of info keys to values.
`validationStringency`	string	`silent`	Controls the behavior when parsing a malformed row. If `silent`, the row will be dropped silently. If `lenient`, the row will be dropped and a warning will be logged. If `strict`, an exception will be thrown and reading will fail.

Note: Starting from Glow 0.4.0, the `splitToBiallelic` option for the VCF reader no longer exists. To split multiallelic variants to biallelics use the *split_multiallelics* transformer after loading the VCF.

Note: Glow includes a VCF reader that uses htsjdk for initial parsing as well as a reader that parses VCF lines to Spark rows directly.

As of release 1.0.0, the direct reader is enabled by default. To use the htsjdk based reader, set the Spark config `io.projectglow.vcf.fastReaderEnabled` to `false`.

Important: The VCF reader uses the 0-start, half-open (zero-based) coordinate system. This means that the `start` values in the DataFrame will be 1 lower than the values that appear in the VCF file. For instance, if a variant has a POS value of 10 in a VCF file, the `start` column in the DataFrame will contain the value 9. When writing to a VCF file, Glow converts positions back to a 1-based coordinate system as required by the VCF specification.

You can save a DataFrame as a VCF file, which you can then read with other tools. To write a DataFrame as a single VCF file specify the format `"bigvcf"`:

```
df.write.format("bigvcf").save(path)
```

The file extension of the output path determines which, if any, compression codec should be used. For instance, writing to a path such as `/genomics/my_vcf.vcf.bgz` will cause the output file to be block gzipped.

If you'd rather save a sharded VCF where each partition saves to a separate file:

```
df.write.format("vcf").save(path)
```

To control the behavior of the sharded VCF writer, you can provide the following option:

Parameter	Type	Default	Description
`compression`	string	n/a	A compression codec to use for the output VCF file. The value can be the full name of a compression codec class (for example `GzipCodec`) or a short alias (for example `gzip`). To use the block gzip codec, specify `bgzf`.

For both the single and sharded VCF writer, you can use the following options:

Parameter	Type	Default	Description
`vcfHeader`	string	`infer`	If `infer`, infers the header from the DataFrame schema. This value can be a complete header starting with `##` or a Hadoop filesystem path to a VCF file. The header from this file is used as the VCF header for each partition.
`validationStringency`	string	`silent`	Controls the behavior when parsing a malformed row. If `silent`, the row will be dropped silently. If `lenient`, the row will be dropped and a warning will be logged. If `strict`, an exception will be thrown and writing will fail.

If the header is inferred from the DataFrame, the sample IDs are derived from the rows. If the sample IDs are missing, they will be represented as `sample_n`, for which `n` reflects the index of the sample in a row. In this case, there must be the same number of samples in each row.

- For the big VCF writer, the inferred sample IDs are the distinct set of all sample IDs from the DataFrame.
- For the sharded VCF writer, the sample IDs are inferred from the first row of each partition and must be the same for each row. If the rows do not contain the same samples, provide a complete header of a filesystem path to a VCF file.

3.2.2 BGEN

Glow provides the ability to read BGEN files, including those distributed by the UK Biobank project.

```
df = spark.read.format("bgen").load(path)
```

As with the VCF reader, the provided path can be a file, directory, or glob pattern. If `.bgi` index files are located in the same directory as the data files, the reader uses the indexes to more efficiently traverse the data files. Data files can be processed even if indexes do not exist. The schema of the resulting DataFrame matches that of the VCF reader.

Parameter	Type	Default	Description
`useBgenIndex`	boolean	`true`	If true, use `.bgi` index files.
`sampleFilePath`	string	n/a	Path to a `.sample` Oxford sample information file containing sample IDs if not stored in the BGEN file.
`sampleIdColumn`	string	`ID_2`	Name of the column in the `.sample` file corresponding to the sample IDs.
`emitHardCalls`	boolean	`true`	If true, adds genotype calls for diploids based on the posterior probabilities.
`hardCallThreshold`	double	0.9	Sets the threshold for hard calls.

Important: The BGEN reader and writer assume that the first allele in the `.bgen` file is the reference allele, and that all following alleles are alternate alleles.

You can use the `DataFrameWriter` API to save a single BGEN file, which you can then read with other tools.

```
df.write.format("bigbgen").save(path)
```

If the genotype arrays are missing ploidy and/or phasing information, the BGEN writer infers the values using the provided values for ploidy, phasing, or `posteriorProbabilities` in the genotype arrays. You can provide the value for ploidy using an integer value `ploidy` or it can be inferred using the length of an array `calls`, and you can provide the phasing information using a boolean value `phased`.

To control the behavior of the BGEN writer, you can provide the following options:

Parameter	Type	Default	Description
`bitsPerProbability`	integer	16	Number of bits used to represent each probability value. Must be 8, 16, or 32.
`maximumInferredPloidy`	integer	10	The maximum ploidy that will be inferred for unphased data if ploidy is missing.
`defaultInferredPloidy`	integer	2	The inferred ploidy if phasing and ploidy are missing, or ploidy is missing and cannot be inferred from `posteriorProbabilities`.
`defaultInferredPhasing`	boolean	`false`	The inferred phasing if phasing is missing and cannot be inferred from `posteriorProbabilities`.

3.2.3 PLINK

Glow provides the ability to read binary PLINK binary PED (BED) files with accompanying BIM and FAM files. The provided path can be a file or glob pattern.

```
df = spark.read.format("plink").load("{prefix}.bed".format(prefix=prefix))
```

The schema of the resulting DataFrame matches that of the VCF reader. The accompanying variant and sample information files must be located at `{prefix}.bim` and `{prefix}.fam`.

Parameter	Type	Default	Description
`includeSampleIds`	boolean	`true`	If true, each genotype includes the name of the sample ID it belongs to.
`bimDelimiter`	string	(tab)	Whitespace delimiter in the `{prefix}.bim` file.
`famDelimiter`	string	(space)	Whitespace delimiter in the `{prefix}.fam` file.
`mergeFidIid`	boolean	`true`	If true, sets the sample ID to the family ID and individual ID merged with an underscore delimiter. If false, sets the sample ID to the individual ID.

Important: The PLINK reader sets the first allele in the `.bed` file as the alternate allele, and the second allele as the reference allele.

Notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/variant-data.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/variant-data.html" id='1959205081541642431' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.3 Read Genome Annotations (GFF3) as a Spark DataFrame

GFF3 (Generic Feature Format Version 3) is a 9-column tab-separated text file format commonly used to store genomic annotations. Typically, the majority of annotation data in this format appears in the ninth column, called `attributes`, as a semi-colon-separated list of `<tag>=<value>` entries. If Spark's standard `csv` data source is used to read GFF3 files, the whole list of attribute tag-value pairs will be read as a single string-typed column, making queries on these tags/values cumbersome.

To address this issue, Glow provides the `gff` data source. In addition to loading the first 8 columns of GFF3 as properly typed columns, the `gff` data source is able to parse all attribute tag-value pairs in the ninth column of GFF3 and create an appropriately typed column for each tag. In each row, the column corresponding to a tag will contain the tag's value in that row (or `null` if the tag does not appear in the row).

Like any Spark data source, reading GFF3 files using the `gff` data source can be done in a single line of code:

```
df = spark.read.format("gff").load(path)
```

The `gff` data source supports all compression formats supported by Spark's `csv` data source, including `.gz` and `.bgz` files. It also supports reading globs of files in one command.

Note: The `gff` data source ignores any comment and directive lines (lines starting with `#`) in the GFF3 file as well as any FASTA lines that may appear at the end of the file.

3.3.1 Schema

1. Inferred schema

If no user-specified schema is provided (as in the example above), the data source infers the schema as follows:

- The first 8 fields of the schema ("base" fields) correspond to the first 8 columns of the GFF3 file. Their names, types and order will be as shown below:

```
|-- seqId: string (nullable = true)
|-- source: string (nullable = true)
|-- type: string (nullable = true)
|-- start: long (nullable = true)
|-- end: long (nullable = true)
|-- score: double (nullable = true)
|-- strand: string (nullable = true)
|-- phase: integer (nullable = true)
```

Note: Although the `start` column in the GFF3 file is 1-based, the `start` field in the DataFrame will be 0-based to match the general practice in Glow.

- The next fields in the inferred schema will be created as the result of parsing the `attributes` column of the GFF3 file. Each tag will have its own field in the schema. Fields corresponding to any "official" tag (those referred to as tags with pre-defined meaning) come first, followed by fields corresponding to any other tag ("unofficial" tags) in alphabetical order.

  The complete list of official fields, their data types, and order are as shown below:

```
|-- ID: string (nullable = true)
|-- Name: string (nullable = true)
|-- Alias: string (nullable = true)
|-- Parent: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- Target: string (nullable = true)
|-- Gap: string (nullable = true)
|-- DerivesFrom: string (nullable = true)
|-- Note: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- Dbxref: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- OntologyTerm: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- Is_circular: boolean (nullable = true)
```

  The unofficial fields will be of `string` type.

Note:

  - If any of official tags does not appear in any row of the GFF3 file, the corresponding field will be excluded from the inferred schema.
  - The official/unofficial field name will be exactly as the corresponding tag appears in the GFF3 file (in terms of letter case).
  - The parser is insensitive to the letter case of the tag, e.g., if the `attributes` column in the GFF3 file contains both `note` and `Note` tags, they will be both mapped to the same column in the DataFrame. The name of the column in this case will be either `note` or `Note`, chosen randomly.

2. User-specified schema

As with any Spark data source, the `gff` data source is also able to accept a user-specified schema through the `.schema` command. The user-specified schema can have any subset of the base, official, and unofficial fields. The data source is able to read only the specified base fields and parse out only the specified official and unofficial fields from the `attributes` column of the GFF3 file. Here is an example of how the user can specify some base, official, and unofficial fields while reading the GFF3 file:

```
mySchema = StructType(
  [StructField('seqId', StringType()),              # Base field
   StructField('start', LongType()),                # Base field
   StructField('end', LongType()),                  # Base field
   StructField('ID', StringType()),                 # Official field
   StructField('Dbxref', ArrayType(StringType())),  # Official field
   StructField('mol_type', StringType())]           # Unofficial field
)

df_user_specified = spark.read.format("gff").schema(mySchema).load(path)
```

Note:

- The base field names in the user-specified schema must match the names in the *list above* in a case-sensitive manner.
- The official and unofficial fields will be matched with their corresponding tags in the GFF3 file in a case-and-underscore-insensitive manner. For example, if the GFF3 file contains the official tag `db_xref`, a user-specified schema field with the name `dbxref`, `Db_Xref`, or any other case-and-underscore-insensitive match will correspond to that tag.
- The user can also include the original `attributes` column of the GFF3 file as a string field by including `StructField('attributes', StringType())` in the schema.

Notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/gff.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/gff.html" id='2870708909679252959' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.4 Create a Genomics Delta Lake

Genomics data is usually stored in specialized flat-file formats such as VCF or BGEN.

The example below shows how to ingest a VCF into a genomics Delta Lake table using Glow in Python (R, Scala, and SQL are also supported).

You can use Delta tables for second-latency queries, performant range-joins (similar to the single-node bioinformatics tool bedtools intersect), aggregate analyses such as calculating summary statistics, machine learning or deep learning.

Tip: We recommend ingesting VCF files into Delta tables once volumes reach >1000 samples, >10 billion genotypes or >1 terabyte.

3.4.1 VCF to Delta Lake table notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/vcf2delta.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/vcf2delta.html" id='4973926517334243532' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.5 Variant Quality Control

Glow includes a variety of tools for variant quality control.

Tip: This topic uses the terms “variant” or “variant data” to refer to single nucleotide variants and short indels.

You can calculate quality control statistics on your variant data using Spark SQL functions, which can be expressed in Python, R, Scala, or SQL.

Function	Arguments	Return
`hardy_weinberg`	The `genotypes` array. This function assumes that the variant has been converted to a biallelic representation.	A struct with two elements: the expected heterozygous frequency according to Hardy-Weinberg equilibrium and the associated p-value.
`call_summary_stats`	The `genotypes` array	A struct containing the following summary stats: • `callRate`: The fraction of samples with a called genotype • `nCalled`: The number of samples with a called genotype • `nUncalled`: The number of samples with a missing or uncalled genotype, as represented by a '.' in a VCF or -1 in a DataFrame. • `nHet`: The number of heterozygous samples • `nHomozygous`: An array with the number of samples that are homozygous for each allele. The 0th element describes how many sample are hom-ref. • `nNonRef`: The number of samples that are not hom-ref • `nAllelesCalled`: An array with the number of times each allele was seen • `alleleFrequencies`: An array with the frequency for each allele
`dp_summary_stats`	The `genotypes` array	A struct containing the min, max, mean, and sample standard deviation for genotype depth (DP in VCF v4.2 specificiation) across all samples
`gq_summary_stats`	The `genotypes` array	A struct containing the min, max, mean, and sample standard deviation for genotype quality (GQ in VCF v4.2 specification) across all samples

3.5.1 Notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/variant-qc-demo.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/variant-qc-demo.html" id='-546301641495193183' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.6 Sample Quality Control

You can calculate quality control statistics on your variant data using Spark SQL functions, which can be expressed in Python, R, Scala, or SQL.

Each of these functions returns an array of structs containing metrics for one sample. If sample ids are including in the input DataFrame, they will be propagated to the output. The functions assume that the genotypes in each row of the input DataFrame contain the same samples in the same order.

Functions	Arguments	Return
`sample_call_summary_stats`	`referenceAllele` string, `alternateAlleles` array of strings, `genotypes` array `calls`	A struct containing the following summary stats: • `callRate`: The fraction of variants where this sample has a called genotype. Equivalent to `nCalled / (nCalled + nUncalled)` • `nCalled`: The number of variants where this sample has a called genotype • `nUncalled`: The number of variants where this sample does not have a called genotype • `nHomRef`: The number of variants where this sample is homozygous reference • `nHet`: The number of variants where this sample is heterozygous • `nHomVar`: The number of variants where this sample is homozygous non reference • `nSnv`: The number of calls where this sample has a single nucleotide variant. This value is the sum of `nTransition` and `nTransversion` • `nInsertion`: Insertion variant count • `nDeletion`: Deletion variant count • `nTransition`: Transition count • `nTransversion`: Transversion count • `nSpanningDeletion`: The number of calls where this sample has a spanning deletion • `rTiTv`: Ratio of transitions to tranversions (`nTransition / nTransversion`) • `rInsertionDeletion`: Ratio of insertions to deletions (`nInsertion / nDeletion`) • `rHetHomVar`: Ratio of heterozygous to homozygous variant calls (`nHet / nHomVar`)
`sample_dp_summary_stats`	`genotypes` array with a `depth` field	A struct with `min`, `max`, `mean`, and `stddev`
`sample_gq_summary_stats`	`genotypes` array with a `conditionalQuality` field	A struct with `min`, `max`, `mean`, and `stddev`

3.6.1 Computing user-defined sample QC metrics

In addition to the built-in QC functions discussed above, Glow provides two ways to compute user-defined per-sample statistics.

`aggregate_by_index`

First, you can aggregate over each sample in a genotypes array using the `aggregate_by_index` function.

`aggregate_by_index(array, initial_state, update_function, merge_function, eval_function)`

Name	Type	Description
`array`	`array<T>`	An array-typed column. There are no requirements on the element datatype. This array is expected to be the same length for each row in the input DataFrame. The output of `aggregate_by_index` is an array with the same length as each input row.
`initial_state`	`U`	The initial aggregation state for each sample.
`update_function`	`<U, T> -> U`	A function that returns a new single sample aggregation state given the current aggregation state and a new data element.
`merge_function`	`<U, U> -> U`	A function that combines two single sample aggregation states. This function is necessary since the aggregation is computed in a distributed manner across all nodes in the cluster.
`eval_function` (optional)	`U -> V`	A function that returns the output for a sample given that sample's aggregation state. This function is optional. If it is not specified, the aggregation state will be returned.

For example, this code snippet uses `aggregate_by_index` to compute the mean for each array position:

```
aggregate_by_index(
  array_col,
  (0d, 0l),
  (state, element) -> (state.col1 + element, state.col2 + 1),
  (state1, state2) -> (state1.col1 + state2.col1, state1.col2 + state2.col2),
  state -> state.col1 / state.col2)
```

Explode and aggregate

If your dataset is not in a normalized, pVCF-esque shape, or if you want the aggregation output in a table rather than a single array, you can explode the `genotypes` array and use any of the aggregation functions built into Spark. For example, this code snippet computes the number of sites with a non-reference allele for each sample:

```
import pyspark.sql.functions as fx
exploded_df = df.withColumn("genotype", fx.explode("genotypes"))\
  .withColumn("hasNonRef", fx.expr("exists(genotype.calls, call -> call != -1 and call !
↪= 0)"))

agg = exploded_df.groupBy("genotype.sampleId", "hasNonRef")\
  .agg(fx.count(fx.lit(1)))\
  .orderBy("sampleId", "hasNonRef")
```

Notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/sample-qc-demo.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/sample-qc-demo.html" id='7956224706607613904' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.7 Liftover

LiftOver converts genomic data between reference assemblies. The UCSC liftOver tool uses a chain file to perform simple coordinate conversion, for example on BED files. The Picard LiftOverVcf tool also uses the new reference assembly file to transform variant information (eg. alleles and INFO fields). Glow can be used to run *coordinate liftOver* and *variant liftOver*.

3.7.1 Create a liftOver cluster

For both coordinate and variant liftOver, you need a chain file on every node of the cluster. On a Databricks cluster, an example of a cluster-scoped init script you can use to download the required file for liftOver from the b37 to the hg38 reference assembly is as follows:

```
#!/usr/bin/env bash
set -ex
set -o pipefail
mkdir /opt/liftover
curl https://raw.githubusercontent.com/broadinstitute/gatk/master/scripts/funcotator/
↪data_sources/gnomAD/b37ToHg38.over.chain --output /opt/liftover/b37ToHg38.over.chain
```

3.7.2 Coordinate liftOver

To perform liftOver for genomic coordinates, use the function `lift_over_coordinates`. `lift_over_coordinates` has the following parameters.

- chromosome: `string`
- start: `long`
- end: `long`
- chain file: `string` (constant value, such as one created with `lit()`)
- minimum fraction of bases that must remap: `double` (optional, defaults to `.95`)

The returned `struct` has the following values if liftOver succeeded. If not, the function returns `null`.

- `contigName`: `string`
- `start`: `long`
- `end`: `long`

```
output_df = input_df.withColumn('lifted', glow.lift_over_coordinates('contigName', 'start
↪',
  'end', chain_file, 0.99))
```

3.7.3 Variant liftOver

For genetic variant data, use the `lift_over_variants` transformer. In addition to performing liftOver for genetic coordinates, variant liftOver performs the following transformations:

- Reverse-complement and left-align the variant if needed
- Adjust the SNP, and correct allele-frequency-like INFO fields and the relevant genotypes if the reference and alternate alleles have been swapped in the new genome build

Pull a target assembly reference file down to every node in the Spark cluster in addition to a chain file before performing variant liftOver.

The `lift_over_variants` transformer operates on a DataFrame containing genetic variants and supports the following options:

Parameter	Default	Description
`chain_file`	n/a	The path of the chain file.
`reference_file`	n/a	The path of the target reference file.
`min_match_ratio`	.95	Minimum fraction of bases that must remap.

The output DataFrame's schema consists of the input DataFrame's schema with the following fields appended:

- `INFO_SwappedAlleles`: `boolean` (null if liftOver failed, true if the reference and alternate alleles were swapped, false otherwise)
- `INFO_ReverseComplementedAlleles`: `boolean` (null if liftover failed, true if the reference and alternate alleles were reverse complemented, false otherwise)
- `liftOverStatus`: `struct`
  - `success`: `boolean` (true if liftOver succeeded, false otherwise)
  - `errorMessage`: `string` (null if liftOver succeeded, message describing reason for liftOver failure otherwise)

If liftOver succeeds, the output row contains the liftOver result and `liftOverStatus.success` is true. If liftOver fails, the output row contains the original input row, the additional `INFO` fields are null, `liftOverStatus.success` is false, and `liftOverStatus.errorMessage` contains the reason liftOver failed.

```
output_df = glow.transform('lift_over_variants', input_df, chain_file=chain_file,␣
↪reference_file=reference_file)
```

Liftover notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/lift-over.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/lift-over.html" id='-4929768015519559733' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.8 Variant Normalization

Different genomic analysis tools often represent the same genomic variant in different ways, making it non-trivial to integrate and compare variants across call sets. Therefore, **variant normalization** is an essential step to be applied on variants before further downstream analysis to make sure the same variant is represented identically in different call sets. Normalization is achieved by making sure the variant is parsimonious and left-aligned (see Variant Normalization for more details).

Glow provides variant normalization capabilities as a DataFrame transformer as well as a SQL expression function with a Python API, bringing unprecedented scalability to this operation.

Note: Glow's variant normalization algorithm follows the same logic as those used in normalization tools such as bcftools norm and vt normalize. This normalization logic is different from the one used by GATK's LeftAlignAndTrimVariants, which sometimes yields incorrect normalization (see Variant Normalization for more details).

3.8.1 `normalize_variants` Transformer

The `normalize_variants` transformer can be applied to normalize a variant DataFrame, such as one generated by loading VCF or BGEN files. The output of the transformer is described under the `replace_columns` option below.

3.8.2 Usage

Assuming `df_original` is a variable of type DataFrame which contains the genomic variant records, and `ref_genome_path` is a variable of type String containing the path to the reference genome file, a minimal example of using this transformer for normalization is as follows:

Python

Scala

```
df_normalized = glow.transform("normalize_variants", df_original, reference_genome_
↪path=ref_genome_path)
```

```
df_normalized = Glow.transform("normalize_variants", df_original, Map("reference_genome_
↪path" -> ref_genome_path))
```

3.8.3 Options

The `normalize_variants` transformer has the following options:

Option	Type	Possible values and description
`reference_genome_path`	string	Path to the reference genome `.fasta` or `.fa` file. This file must be accompanied with a `.fai` index file in the same folder.
`replace_columns`	boolean	`False`: The transformer does not modify the original `start`, `end`, `referenceAllele` and `alternateAlleles` columns. Instead, a StructType column called `normalizationResult` is added to the DataFrame. This column contains the normalized `start`, `end`, `referenceAllele`, and `alternateAlleles` columns as well as the `normalizationStatus` StructType as the fifth field, which contains the following subfields: `changed`: Indicates whether the variant data was changed as a result of normalization `errorMessage`: An error message in case the attempt at normalizing the row hit an error. In this case, the `changed` field will be set to `False`. If no errors occur this field will be `null`. In case of error, the first four fields in `normalizationResult` will be `null`. `True` (default): The original `start`, `end`, `referenceAllele`, and `alternateAlleles` columns are replaced with the normalized values in case they have changed. Otherwise (in case of no change or an error), the original `start`, `end`, `referenceAllele`, and `alternateAlleles` are not modified. A StructType `normalizationStatus` column is added to the DataFrame with the same subfields explained above.
`mode` (deprecated)	string	`normalize`: Only normalizes the variants (if user does not pass the option, `normalize` is assumed as default) `split_and_normalize`: Split multiallelic variants to biallelic variants and then normalize the variants. This usage is deprecated.

3.8.4 `normalize_variant` Function

The normalizer can also be used as a SQL expression function. See *Glow PySpark Functions* for details on how to use it in the Python API. An example of an expression using the `normalize_variant` function is as follows:

```
from pyspark.sql.functions import lit
normalization_expr = glow.normalize_variant('contigName', 'start', 'end',
↪'referenceAllele', 'alternateAlleles', ref_genome_path)
df_normalized = df_original.withColumn('normalizationResult', normalization_expr)
```

Variant normalization notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/normalizevariants.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/normalizevariants.html" id='-8617982944381532667' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.9 Split Multiallelic Variants

Splitting multiallelic variants to biallelic variants is a transformation sometimes required before further downstream analysis. Glow provides the `split_multiallelics` transformer to be applied on a variant DataFrame to split multiallelic variants in the DataFrame to biallelic variants. This transformer is able to handle any number of `ALT` alleles and any ploidy.

Note: The splitting logic used by the `split_multiallelics` transformer is the same as the one used by the vt decompose tool of the vt package with option `-s` (note that the example provided at vt decompose user manual page does not reflect the behavior of `vt decompose -s` completely correctly).

The precise behavior of the `split_multiallelics` transformer is presented below:

- A given multiallelic row with n `ALT` alleles is split to n biallelic rows, each with one of the `ALT` alleles of the original multiallelic row. The `REF` allele in all split rows is the same as the `REF` allele in the multiallelic row.
- Each `INFO` field is appropriately split among split rows if it has the same number of elements as number of `ALT` alleles, otherwise it is repeated in all split rows. The boolean `INFO` field `splitFromMultiAllelic` is added/modified to reflect whether the new row is the result of splitting a multiallelic row through this transformation or not. A new `INFO` field called `OLD_MULTIALLELIC` is added to the DataFrame, which for each split row, holds the `CHROM:POS:REF/ALT` of its original multiallelic row. Note that the `INFO` field must be flattened (as explained *here*) in order to be split by this transformer. Unflattened `INFO` fields (such as those inside an `attributes` field) will not be split, but just repeated in whole across all split rows.
- Genotype fields for each sample are treated as follows: The `GT` field becomes biallelic in each row, where the original `ALT` alleles that are not present in that row are replaced with no call. The fields with number of entries equal to number of `REF` + `ALT` alleles, are properly split into rows, where in each split row, only entries corresponding to the `REF` allele as well as the `ALT` allele present in that row are kept. The fields which follow colex order (e.g., `GL`, `PL`, and `GP`) are properly split between split rows where in each row only the elements corresponding to genotypes comprising of the `REF` and `ALT` alleles in that row are listed. Other genotype fields are just repeated over the split rows.
- Any other field in the DataFrame is just repeated across the split rows.

As an example (shown in VCF file format), the following multiallelic row

```
#CHROM  POS     ID      REF     ALT     QUAL    FILTER  INFO    FORMAT  SAMPLE1
20      101     .       A       ACCA,TCGG       .       PASS    VC=INDEL;AC=3,2;AF=0.375,
↪0.25;AN=8      GT:AD:DP:GQ:PL  0/1:2,15,31:30:99:2407,0,533,697,822,574
```

will be split into the following two biallelic rows:

```
#CHROM  POS     ID      REF     ALT     QUAL    FILTER  INFO    FORMAT  SAMPLE1
20      101     .       A       ACCA    .       PASS    VC=INDEL;AC=3;AF=0.375;AN=8;OLD_
↪MULTIALLELIC=20:101:A/ACCA/TCGG GT:AD:DP:GQ:PL  0/1:2,15:30:99:2407,0,533
20      101     .       A       TCGG    .       PASS    VC=INDEL;AC=2;AF=0.25;AN=8;OLD_
↪MULTIALLELIC=20:101:A/ACCA/TCGG  GT:AD:DP:GQ:PL  0/.:2,31:30:99:2407,697,574
```

3.9.1 Usage

Assuming `df_original` is a variable of type DataFrame which contains the genomic variant records, an example of using this transformer for splitting multiallelic variants is:

Python

Scala

```
df_split = glow.transform("split_multiallelics", df_original)
```

```
df_split = Glow.transform("split_multiallelics", df_original)
```

Tip: The `split_multiallelics` transformer is often significantly faster if the *whole-stage code generation* feature of Spark Sql is turned off. Therefore, it is recommended that you temporarily turn off this feature using the following command before using this transformer.

Python

Scala

```
spark.conf.set("spark.sql.codegen.wholeStage", False)
```

```
spark.conf.set("spark.sql.codegen.wholeStage", false)
```

Remember to turn this feature back on after your split DataFrame is materialized.

Split Multiallelic Variants notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/splitmultiallelics-transformer.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/splitmultiallelics-transformer.html" id='-1556890171916256631' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.10 Merging Variant Datasets

You can use Glow and Spark to merge genomic variant datasets from non-overlapping sample sets into a multi-sample dataset. In these examples, we will read from VCF files, but the same logic works on *DataFrames backed by other file formats*.

First, read the VCF files into a single Spark DataFrame:

```
from pyspark.sql.functions import *

df = spark.read.format('vcf').load([path1, path2])

# Alternatively, you can use the "union" DataFrame method if the VCF files have the same␣
↪schema
df1 = spark.read.format('vcf').load(path1)
df2 = spark.read.format('vcf').load(path2)
df = df1.union(df2)
```

The resulting DataFrame contains all records from the VCFs you want to merge, but the genotypes from different samples at the same site have not been combined. You can use an aggregation to combine the genotype arrays.

```
from pyspark.sql.functions import *

merged_df = df.groupBy('contigName', 'start', 'end', 'referenceAllele', 'alternateAlleles
↪')\
  .agg(sort_array(flatten(collect_list('genotypes'))).alias('genotypes'))
```

Important: When reading VCF files for a merge operation, sampleId must be the first field in the genotype struct. This is the default Glow schema.

The genotypes from different samples now appear in the same genotypes array.

Note: If the VCFs you are merging contain different sites, elements will be missing from the genotypes array after aggregation. Glow automatically fills in missing genotypes when writing to bigvcf, so an exported VCF will still contain all samples.

3.10.1 Aggregating INFO fields

To preserve INFO fields in a merge, you can use the aggregation functions in Spark. For instance, to emit an INFO_DP column that is the sum of the INFO_DP columns across all samples:

```
from pyspark.sql.functions import *

merged_df = df.groupBy('contigName', 'start', 'end', 'referenceAllele', 'alternateAlleles
↪')\
  .agg(sort_array(flatten(collect_list('genotypes'))).alias('genotypes'),
       sum('INFO_DP').alias('INFO_DP'))
```

3.10.2 Joint genotyping

The merge logic in this document allows you to quickly aggregate genotyping array data or single sample VCFs. For a more sophisticated aggregation that unifies alleles at overlapping sites and uses cohort-level statistics to refine genotype calls, we recommend running a joint genotyping pipeline.

Notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/merge-vcf.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/merge-vcf.html" id='1450444527845052259' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.11 Hail Interoperation

Glow includes functionality to enable conversion between a Hail MatrixTable and a Spark DataFrame, similar to one created with the *native Glow datasources*.

3.11.1 Create a Hail cluster

To use the Hail interoperation functions, you need Hail to be installed on the cluster. On a Databricks cluster, install Hail with an environment variable. See the Hail installation documentation to install Hail in other setups.

3.11.2 Convert to a Glow DataFrame

Convert from a Hail MatrixTable to a Glow-compatible DataFrame with the function `from_matrix_table`.

```
from glow.hail import functions
df = functions.from_matrix_table(mt, include_sample_ids=True)
```

By default, the genotypes contain sample IDs. To remove the sample IDs, set the parameter `include_sample_ids=False`.

3.11.3 Schema mapping

The Glow DataFrame variant fields are derived from the Hail MatrixTable row fields.

Required	Glow DataFrame variant field	Hail MatrixTable row field
Yes	`contigName`	`locus.contig`
Yes	`start`	`locus.position - 1`
Yes	`end`	`info.END` or `locus.position - 1 + len(alleles[0])`
Yes	`referenceAllele`	`alleles[0]`
No	`alternateAlleles`	`alleles[1:]`
No	`names`	`[rsid, varid]`
No	`qual`	`qual`
No	`filters`	`filters`
No	`INFO_<ANY_FIELD>`	`info.<ANY_FIELD>`

The Glow DataFrame genotype sample IDs are derived from the Hail MatrixTable column fields.

All of the other Glow DataFrame genotype fields are derived from the Hail MatrixTable entry fields.

Glow DataFrame genotype field	Hail MatrixTable entry field
`phased`	`GT.phased`
`calls`	`GT.alleles`
`depth`	`DP`
`filters`	`FT`
`genotypeLikelihoods`	`GL`
`phredLikelihoods`	`PL`
`posteriorProbabilities`	`GP`
`conditionalQuality`	`GQ`
`haplotypeQualities`	`HQ`
`expectedAlleleCounts`	`EC`
`mappingQuality`	`MQ`
`alleleDepths`	`AD`
`<ANY_FIELD>`	`<ANY_FIELD>`

Hail interoperation notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/etl/hail-interoperation.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/etl/hail-interoperation.html" id='1060081505490354532' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

3.12 Utility Functions

Glow includes a variety of utility functions for performing basic data manipulation.

3.12.1 Struct transformations

Glow's struct transformation functions change the schema structure of the DataFrame. These transformations integrate with functions whose parameter structs require a certain schema.

- `subset_struct`: subset fields from a struct

```
from pyspark.sql import Row
row_one = Row(Row(str_col='foo', int_col=1, bool_col=True))
row_two = Row(Row(str_col='bar', int_col=2, bool_col=False))
base_df = spark.createDataFrame([row_one, row_two], schema=['base_col'])
subsetted_df = base_df.select(glow.subset_struct('base_col', 'str_col', 'bool_col').
↪alias('subsetted_col'))
```

- `add_struct_fields`: append fields to a struct

```
from pyspark.sql.functions import lit, reverse
added_df = base_df.select(glow.add_struct_fields('base_col', lit('float_col'), lit(3.14),
↪ lit('rev_str_col'), reverse(base_df.base_col.str_col)).alias('added_col'))
```

- expand_struct: explode a struct into columns

```
expanded_df = base_df.select(glow.expand_struct('base_col'))
```

3.12.2 Spark ML transformations

Glow supports transformations between double arrays and Spark ML vectors for integration with machine learning libraries such as Spark's machine learning library (MLlib).

- array_to_dense_vector: transform from an array to a dense vector

```
array_df = spark.createDataFrame([Row([1.0, 2.0, 3.0]), Row([4.1, 5.1, 6.1])], schema=[
↪'array_col'])
dense_df = array_df.select(glow.array_to_dense_vector('array_col').alias('dense_vector_
↪col'))
```

- array_to_sparse_vector: transform from an array to a sparse vector

```
sparse_df = array_df.select(glow.array_to_sparse_vector('array_col').alias('sparse_
↪vector_col'))
```

- vector_to_array: transform from a vector to a double array

```
from pyspark.ml.linalg import SparseVector
row_one = Row(vector_col=SparseVector(3, [0, 2], [1.0, 3.0]))
row_two = Row(vector_col=SparseVector(3, [1], [1.0]))
vector_df = spark.createDataFrame([row_one, row_two])
array_df = vector_df.select(glow.vector_to_array('vector_col').alias('array_col'))
```

- explode_matrix: explode a Spark ML matrix such that each row becomes an array of doubles

```
from pyspark.ml.linalg import DenseMatrix
matrix_df = spark.createDataFrame(Row([DenseMatrix(2, 3, range(6))]), schema=['matrix_col
↪'])
array_df = matrix_df.select(glow.explode_matrix('matrix_col').alias('array_col'))
```

3.12.3 Variant data transformations

Glow supports numeric transformations on variant data for downstream calculations, such as GWAS.

- genotype_states: create a numeric representation for each sample's genotype data. This calculates the sum of the calls (or -1 if any calls are missing); the sum is equivalent to the number of alternate alleles for biallelic variants.

```
from pyspark.sql.types import *

missing_and_hom_ref = Row([Row(calls=[-1,0]), Row(calls=[0,0])])
het_and_hom_alt = Row([Row(calls=[0,1]), Row(calls=[1,1])])
calls_schema = StructField('calls', ArrayType(IntegerType()))
genotypes_schema = StructField('genotypes_col', ArrayType(StructType([calls_schema])))
genotypes_df = spark.createDataFrame([missing_and_hom_ref, het_and_hom_alt],␣
↪StructType([genotypes_schema]))
num_alt_alleles_df = genotypes_df.select(glow.genotype_states('genotypes_col').alias(
↪'num_alt_alleles_col'))
```

(continues on next page)

(continued from previous page)

- `hard_calls`: get hard calls from genotype probabilities. These are determined based on the number of alternate alleles for the variant, whether the probabilities are phased (true for haplotypes and false for genotypes), and a call threshold (if not provided, this defaults to `0.9`). If no calls have a probability above the threshold, the call is set to `-1`.

```
unphased_above_threshold = Row(probabilities=[0.0, 0.0, 0.0, 1.0, 0.0, 0.0], num_alts=2,␣
↪phased=False)
phased_below_threshold = Row(probabilities=[0.1, 0.9, 0.8, 0.2], num_alts=1, phased=True)
uncalled_df = spark.createDataFrame([unphased_above_threshold, phased_below_threshold])
hard_calls_df = uncalled_df.select(glow.hard_calls('probabilities', 'num_alts', 'phased',
↪ 0.95).alias('calls'))
```

- `mean_substitute`: substitutes the missing values of a numeric array using the mean of the non-missing values. Any values that are NaN, null or equal to the missing value parameter are considered missing. If all values are missing, they are substituted with the missing value. If the missing value is not provided, this defaults to `-1`.

```
unsubstituted_row = Row(unsubstituted_values=[float('nan'), None, -1.0, 0.0, 1.0, 2.0, 3.
↪0])
unsubstituted_df = spark.createDataFrame([unsubstituted_row])
substituted_df = unsubstituted_df.select(glow.mean_substitute('unsubstituted_values',␣
↪lit(-1.0)).alias('substituted_values'))
```

CHAPTER

FOUR

TERTIARY ANALYSIS

Perform population-scale statistical analyses of genetic variants.

4.1 Parallelizing Command-Line Bioinformatics Tools With the Pipe Transformer

To use single-node tools on massive data sets, Glow includes a utility called the Pipe Transformer to process Spark DataFrames with command-line tools.

4.1.1 Usage

Consider a minimal case with a DataFrame containing a single column of strings. You can use the Pipe Transformer to reverse each of the strings in the input DataFrame using the `rev` Linux command:

Python

Scala

Provide options through the `arg_map` argument or as keyword args.

```
# Create a text-only DataFrame
df = spark.createDataFrame([['foo'], ['bar'], ['baz']], ['value'])
rev_df = glow.transform('pipe', df, cmd=['rev'], input_formatter='text', output_
↪formatter='text')
```

Provide options as a `Map[String, Any]`.

```
Glow.transform("pipe", df, Map(
    "cmd" -> Seq("grep", "-v", "#INFO"),
    "inputFormatter" -> "vcf",
    "outputFormatter" -> "vcf",
    "inVcfHeader" -> "infer")
)
```

The options in this example demonstrate how to control the basic behavior of the transformer:

- `cmd` is a JSON-encoded array that contains the command to invoke the program
- `input_formatter` defines how each input row should be passed to the program
- `output_formatter` defines how the program output should be converted into a new DataFrame

The input DataFrame can come from any Spark data source — Delta, Parquet, VCF, BGEN, and so on.

4.1.2 Integrating with bioinformatics tools

To integrate with tools for genomic data, you can configure the Pipe Transformer to write each partition of the input DataFrame as VCF by choosing `vcf` as the input and output formatter. Here is an example using bedtools. For a more complex example using The Variant Effect Predictor (VEP) see the notebook example below. Note that the bioinformatics tool must be installed on each virtual machine of the Spark cluster.

```
df = spark.read.format("vcf").load(path)

intersection_df = glow.transform(
    'pipe',
    df,
    cmd=['bedtools', 'intersect', '-a', 'stdin', '-b', bed, '-header', '-wa'],
    input_formatter='vcf',
    in_vcf_header='infer',
    output_formatter='vcf'
)
```

You must specify a method to determine the VCF header when using the *VCF input formatter*. The option `infer` instructs the Pipe Transformer to derive a VCF header from the DataFrame schema. Alternately, you can provide the header as a blob, or you can point to the filesystem path for an existing VCF file with the correct header.

4.1.3 Options

Option keys and values are always strings. You can specify option names in snake or camel case; for example `inputFormatter`, `input_formatter`, and `InputFormatter` are all equivalent.

Option	Description
`cmd`	The command, specified as an array of strings, to invoke the piped program. The program's stdin receives the formatted contents of the input DataFrame, and the output DataFrame is constructed from its stdout. The stderr stream will appear in the executor logs.
`input_formatter`	Converts the input DataFrame to a format that the piped program understands. Built-in input formatters are `text`, `csv`, and `vcf`.
`output_formatter`	Converts the output of the piped program back into a DataFrame. Built-in output formatters are `text`, `csv`, and `vcf`.
`env_*`	Options beginning with `env_` are interpreted as environment variables. Like other options, the environment variable name is converted to lower snake case. For example, providing the option `env_aniMal=MONKEY` results in an environment variable with key `ani_mal` and value `MONKEY` being provided to the piped program.

Some of the input and output formatters take additional options.

VCF input formatter

Option	Description
`in_vcf_header`	How to determine a VCF header from the input DataFrame. Possible values: • `infer`: Derive a VCF header from the DataFrame schema. The inference behavior matches that of the *sharded VCF writer*. • The complete contents of a VCF header starting with ## • A Hadoop filesystem path to a VCF file. The header from this file is used as the VCF header for each partition.

The CSV input and output formatters accept most of the same options as the CSV data source. You must prefix options to the input formatter with `in_`, and options to the output formatter with `out_`. For example, `in_quote` sets the quote character when writing the input DataFrame to the piped program.

The following options are not supported:

- `path` options are ignored
- The `parserLib` option is ignored. `univocity` is always used as the CSV parsing library.

4.1.4 Cleanup

The pipe transformer uses RDD caching to optimize performance. Spark automatically drops old data partitions in a least-recently-used (LRU) fashion. If you would like to manually clean up the RDDs cached by the pipe transformer instead of waiting for them to fall out of the cache, use the pipe cleanup transformer on any DataFrame. Do not perform cleanup until the pipe transformer results have been materialized, such as by being written to a Delta Lake table.

Python

Scala

```
glow.transform('pipe_cleanup', df)
```

```
Glow.transform("pipe_cleanup", df)
```

Pipe Transformer bedtools example notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/tertiary/pipe-transformer.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/tertiary/pipe-transformer.html" id='1296933811104484145' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

Pipe Transformer Variant Effect Predictor (VEP) example notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/tertiary/pipe-transformer-vep.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/tertiary/pipe-transformer-vep.html" id='2110717504770385098' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

4.2 Using Python Statistics Libraries

This notebook demonstrates how to use pandas user-defined functions (UDFs) to run native Python code with PySpark when working with genomic data.

4.2.1 pandas example notebook

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/tertiary/pandas-lmm.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/tertiary/pandas-lmm.html" id='2153115887729681629' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

4.3 GloWGR: Whole Genome Regression

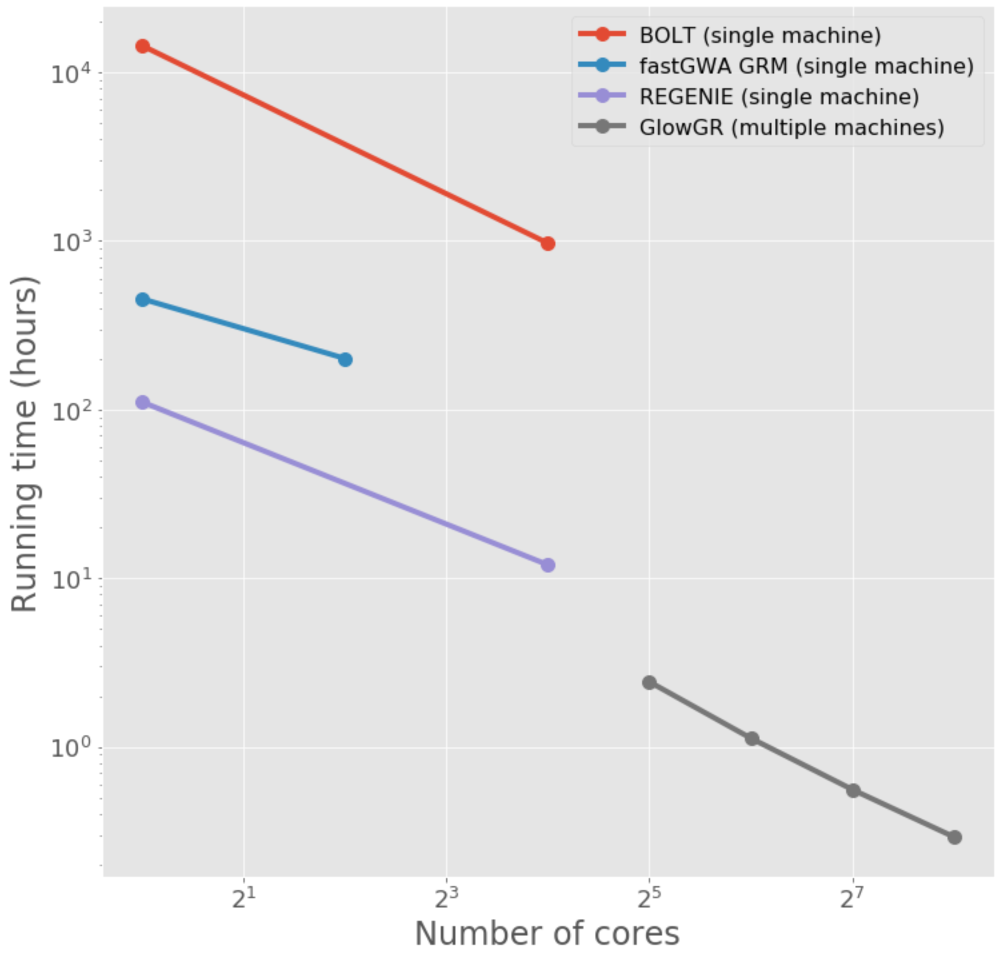
Glow supports Whole Genome Regression (WGR) as GloWGR, a distributed version of the regenie method (see the paper published in Nature Genetics). GloWGR supports two types of phenotypes:

- Quantitative
- Binary

Many steps of the GloWGR workflow explained in this page are common between the two cases. Any step that is different between the two has separate explanations clearly marked by “for quantitative phenotypes” vs. “for binary phenotypes”.

4.3.1 Performance

The following figure demonstrates the performance gain obtained by using parallelized GloWGR in comparision with single machine BOLT, fastGWA GRM, and regenie for fitting WGR models against 50 quantitative phenotypes from the UK Biobank project.

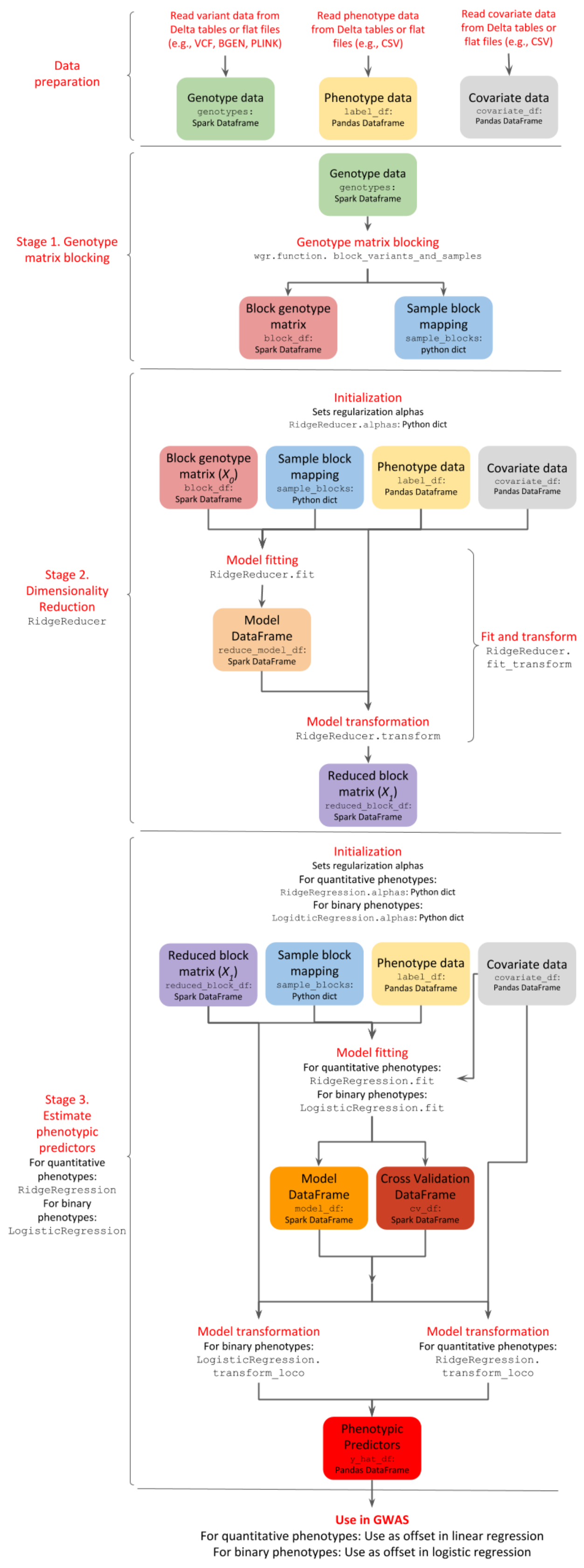


4.3.2 Overview

GloWGR consists of the following stages:

- Block the genotype matrix across samples and variants
- Perform dimensionality reduction with linear ridge regression
- Estimate phenotypic predictors using
  - **For quantitative phenotypes**: linear ridge regression
  - **For binary phenotypes**: logistic ridge regression

The following diagram provides an overview of the operations and data within the stages of GlowWGR and their interrelationship.



Data preparation
Read variant data from Delta tables or flat files (e.g., VCF, BGEN, PLINK)
Read phenotype data from Delta tables or flat files (e.g., CSV)
Read covariate data from Delta tables or flat files (e.g., CSV)
Genotype data genotypes: Spark Dataframe
Phenotype data label_df: Pandas Dataframe
Covariate data covariate_df: Pandas DataFrame
Stage 1. Genotype matrix blocking
Genotype data genotypes: Spark Dataframe
Genotype matrix blocking
wgr.function. block_variants_and_samples
Block genotype matrix block_df: Spark Dataframe
Sample block mapping sample_blocks: python dict
Stage 2. Dimensionality Reduction RidgeReducer
Initialization
Sets regularization alphas
RidgeReducer.alphas: Python dict
Block genotype matrix (X_0) block_df: Spark Dataframe
Sample block mapping sample_blocks: Python dict
Phenotype data label_df: Pandas Dataframe
Covariate data covariate_df: Pandas DataFrame
Model fitting
RidgeReducer.fit
Model DataFrame reduce_model_df: Spark DataFrame
Fit and transform
RidgeReducer. fit_transform
Model transformation
RidgeReducer.transform
Reduced block matrix (X_1) reduced_block_df: Spark DataFrame
Stage 3. Estimate phenotypic predictors
For quantitative phenotypes: RidgeRegression
For binary phenotypes: LogisticRegression
Initialization
Sets regularization alphas
For quantitative phenotypes:
RidgeRegression.alphas: Python dict
For binary phenotypes:
LogidticRegression.alphas: Python dict
Reduced block matrix (X_1) reduced_block_df: Spark DataFrame
Sample block mapping sample_blocks: Python dict
Phenotype data label_df: Pandas Dataframe
Covariate data covariate_df: Pandas DataFrame
Model fitting
For quantitative phenotypes:
RidgeRegression.fit
For binary phenotypes:
LogisticRegression.fit
Model DataFrame model_df: Spark DataFrame
Cross Validation DataFrame cv_df: Spark DataFrame
Model transformation
For binary phenotypes:
LogisticRegression. transform_loco
Model transformation
For quantitative phenotypes:
RidgeRegression. transform_loco
Phenotypic Predictors y_hat_df: Pandas DataFrame
Use in GWAS
For quantitative phenotypes: Use as offset in linear regression
For binary phenotypes: Use as offset in logistic regression

4.3.3 Data preparation

GloWGR accepts three input data components.

1. Genotype data

The genotype data may be read as a Spark DataFrame from any variant data source supported by Glow, such as *VCF, BGEN or PLINK*. For scalability and high-performance repeated use, we recommend storing flat genotype files into *Delta tables*. The DataFrame must include a column `values` containing a numeric representation of each genotype. The genotypic values may not be missing.

When loading the variants into the DataFrame, perform the following transformations:

- Split multiallelic variants with the `split_multiallelics` transformer.
- Create a `values` column by calculating the numeric representation of each genotype. This representation is typically the number of alternate alleles for biallelic variants which can be calculated with `glow.genotype_states`. Replace any missing values with the mean of the non-missing values using `glow.mean_substitute`.

Example

```
from pyspark.sql.functions import col, lit

variants = spark.read.format('vcf').load(genotypes_vcf)
genotypes = variants.withColumn('values', glow.mean_substitute(glow.genotype_states(col(
↪'genotypes'))))
```

2. Phenotype data

The phenotype data is represented as a Pandas DataFrame indexed by the sample ID. Phenotypes are also referred to as labels. Each column represents a single phenotype. It is assumed that there are no missing phenotype values.

- **For quantitative phenotypes:** It is assumed the phenotypes are standardized with zero mean and unit (unbiased) variance.

  Example: Standardization can be performed with Pandas or scikit-learn's StandardScaler.

  ```
  import pandas as pd
  label_df = pd.read_csv(continuous_phenotypes_csv, index_col='sample_id')[[
  ↪'Continuous_Trait_1', 'Continuous_Trait_2']]
  ```

- **For binary phenotypes:** Phenotype values are either 0 or 1. No standardization is needed.

  Example

  ```
  import pandas as pd
  label_df = pd.read_csv(binary_phenotypes_csv, index_col='sample_id')
  ```

3. Covariate data

The covariate data is represented as a Pandas DataFrame indexed by the sample ID. Each column represents a single covariate. It is assumed that there are no missing covariate values, and that the covariates are standardized with zero mean and unit (unbiased) variance.

Example

```
covariate_df = pd.read_csv(covariates_csv, index_col='sample_id')
```

4.3.4 Stage 1. Genotype matrix blocking

The first stage of GloWGR is to generate the block genotype matrix. The `glow.wgr.functions.block_variants_and_samples` function is used for this purpose and creates two objects: a block genotype matrix and a sample block mapping.

Warning: We do not recommend using the `split_multiallelics` transformer and the `block_variants_and_samples` function in the same query due to JVM JIT code size limits during whole-stage code generation. It is best to persist the variants after splitting multiallelics to a Delta table (see *Create a Genomics Delta Lake*) and then read the data from this Delta table to apply `block_variants_and_samples`.

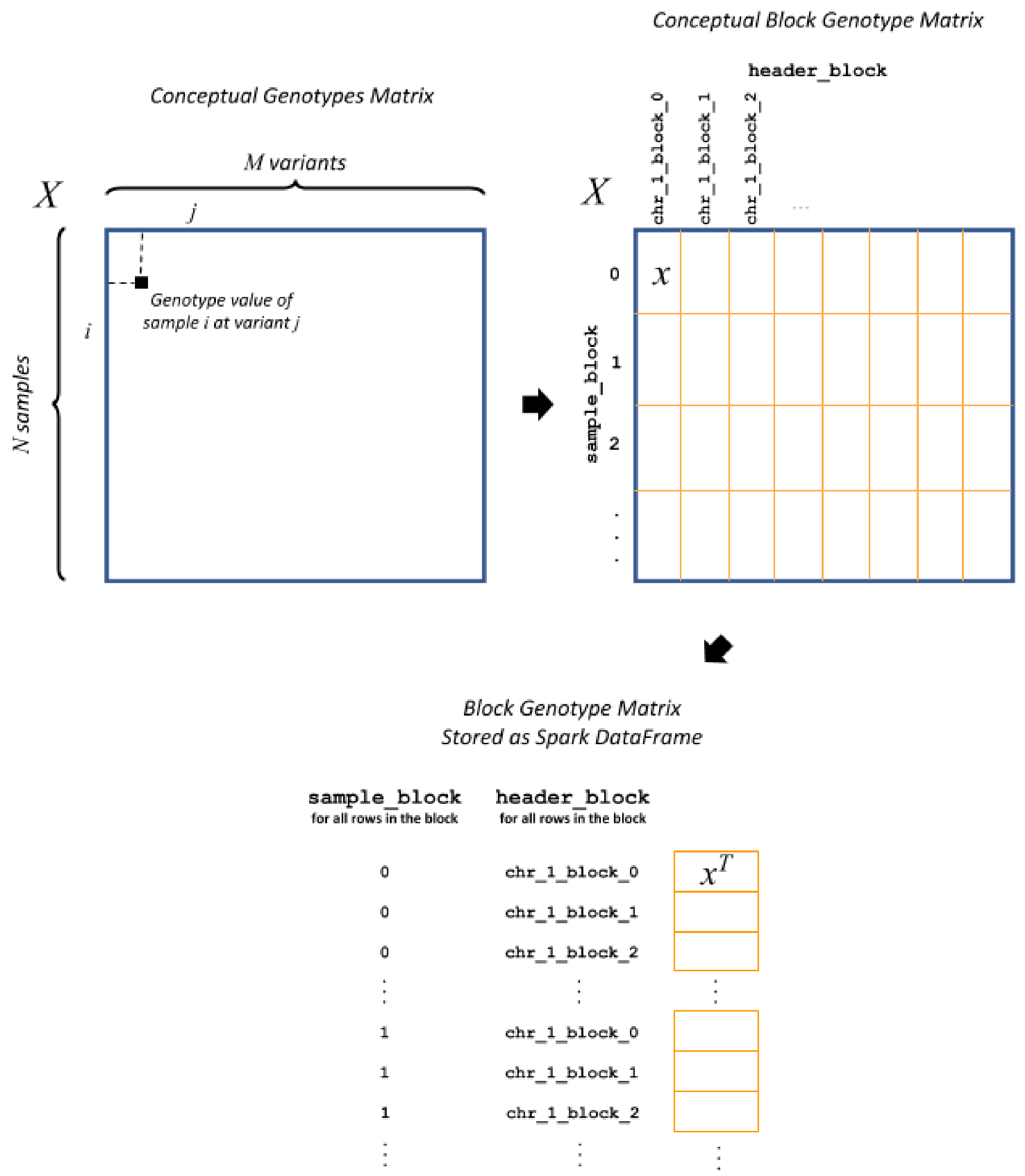
Parameters

- `genotypes`: Genotype DataFrame including the `values` column generated as explained *above*
- `sample_ids`: A python List of sample IDs. Can be created by applying `glow.wgr.functions.get_sample_ids` to a genotype DataFrame
- `variants_per_block`: Number of variants to include in each block. We recommend 1000.
- `sample_block_count`: Number of sample blocks to create. We recommend 10.

Return

The function returns a block genotype matrix and a sample block mapping.

- **Block genotype matrix** (see figure below): The block genotype matrix can be conceptually imagined as an $N \times M$ matrix X where each row represents an individual sample, and each column represents a variant, and each cell (i, j) contains the genotype value for sample i at variant j. Then imagine a coarse grid is laid on top of matrix X such that matrix cells within the same coarse grid cell are all assigned to the same block. Each block x is indexed by a sample block ID (corresponding to a list of rows belonging to the block) and a header block ID (corresponding to a list of columns belonging to the block). The sample block IDs are generally just integers 0 through the number of sample blocks. The header block IDs are strings of the form 'chr_C_block_B', which refers to the Bth block on chromosome C. The Spark DataFrame representing this block matrix can be thought of as the transpose of each block, i.e., x^T, all stacked one atop another. Each row in the DataFrame represents the values from a particular column of X for the samples corresponding to a particular sample block.



The fields in the DataFrame and their content for a given row are as follows:

- `sample_block`: An ID assigned to the block x containing the group of samples represented on this row
- `header_block`: An ID assigned to the block x containing this header
- `header`: The column name in the conceptual genotype matrix X
- `size`: The number of individuals in the sample block
- `values`: Genotype values for the header in this sample block. If the matrix is sparse, contains only non-zero values.
- `position`: An integer assigned to this header that specifies the correct sort order for the headers in this block

  - `mu`: The mean of the genotype values for this header
  - `sig`: The standard deviation of the genotype values for this header

> **Warning:** Variant rows in the input DataFrame whose genotype values are uniform across all samples are filtered from the output block genotype matrix.

- **Sample block mapping**: The sample block mapping is a python dictionary containing key-value pairs, where each key is a sample block ID and each value is a list of sample IDs contained in that sample block. The order of these IDs match the order of the `values` arrays in the block genotype DataFrame.

Example

```
from glow.wgr import RidgeReduction, RidgeRegression, LogisticRidgeRegression, block_
↪variants_and_samples, get_sample_ids
from pyspark.sql.functions import col, lit

variants_per_block = 1000
sample_block_count = 10
sample_ids = get_sample_ids(genotypes)
block_df, sample_blocks = block_variants_and_samples(
    genotypes, sample_ids, variants_per_block, sample_block_count)
```

4.3.5 Stage 2. Dimensionality reduction

Having the block genotype matrix, the first stage is to apply a dimensionality reduction to the block matrix X using the `RidgeReducer`. After `RidgeReducer` is initialized, dimensionality reduction is accomplished within two steps:

1. Model fitting, performed by the `RidgeReducer.fit` function, which fits multiple ridge models within each block x.
2. Model transformation, performed by the `RidgeReducer.transform` function, which produces a new block matrix where each column represents the prediction of one ridge model applied within one block.

This approach to model building is generally referred to as **stacking**. We call the starting block genotype matrix the **level 0** matrix in the stack, denoted by X_0, and the output of the ridge reduction step the **level 1** matrix, denoted by X_1. The `RidgeReducer` class is initialized with a list of ridge regularization values (here referred to as alpha). Since ridge models are indexed by these alpha values, the `RidgeReducer` will generate one ridge model per value of alpha provided, which in turn will produce one column per block in X_0. Therefore, the final dimensions of X_1 for a single phenotype will be $N \times (L \times K)$, where L is the number of header blocks in X_0 and K is the number of alpha values provided to the `RidgeReducer`. In practice, we can estimate a span of alpha values in a reasonable order of magnitude based on guesses at the heritability of the phenotype we are fitting.

1. Initialization

When the `RidgeReducer` is initialized, it assigns names to the provided alphas and stores them in a python dictionary accessible as `RidgeReducer.alphas`. If alpha values are not provided, they will be generated during `RidgeReducer.fit` based on the number of unique headers in the blocked genotype matrix X_0, denoted by h_0, and a set of heritability values. More specifically,

$$\alpha = h_0\big[\frac{1}{0.99}, \frac{1}{0.75}, \frac{1}{0.50}, \frac{1}{0.25}, \frac{1}{0.01}\big]$$

These values are only sensible if the phenotypes are on the scale of one.

Example

```
reduction = RidgeReduction(block_df, label_df, sample_blocks, covariate_df)
```

2. Model fitting

The reduction of a block x_0 from X_0 to the corresponding block x_1 from X_1 is accomplished by the matrix multiplication $x_0 B = x_1$, where B is a coefficient matrix of size $m \times K$, where m is the number of columns in block x_0 and K is the number of alpha values used in the reduction. As an added wrinkle, if the ridge reduction is being performed against multiple phenotypes at once, each phenotype will have its own B, and for convenience we panel these next to each other in the output into a single matrix, so B in that case has dimensions $m \times (K \times P)$ where P is the number of phenotypes. Each matrix B is specific to a particular block in X_0, so the Spark DataFrame produced by the `RidgeReducer` can be thought of matrices B from all the blocks, one stacked atop another.

Parameters

- `block_df`: Spark DataFrame representing the beginning block matrix
- `label_df`: Pandas DataFrame containing the target labels used in fitting the ridge models
- `sample_blocks`: Dictionary containing a mapping of sample block IDs to a list of corresponding sample IDs
- `covariate_df`: Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional)

Return

The fields in the model DataFrame are:

- `header_block`: An ID assigned to the block x_0 to the coefficients in this row
- `sample_block`: An ID assigned to the block x_0 containing the group of samples represented on this row
- `header`: The column name in the conceptual genotype matrix X_0 that corresponds to a particular row in the coefficient matrix B
- `alphas`: List of alpha names corresponding to the columns of B
- `labels`: List of labels (i.e., phenotypes) corresponding to the columns of B
- `coefficients`: List of the actual values from a row in B

Example

```
model_df = reduction.fit()
```

3. Model transformation

After fitting, the `RidgeReducer.transform` method can be used to generate X_1 from X_0.

Parameters

- `block_df`: Spark DataFrame representing the beginning block matrix
- `label_df`: Pandas DataFrame containing the target labels used in fitting the ridge models
- `sample_blocks`: Dictionary containing a mapping of sample block IDs to a list of corresponding sample IDs
- `model_df`: Spark DataFrame produced by the `RidgeReducer.fit` function, representing the reducer model
- `covariate_df`: Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

Return

The output of the transformation is analogous to the block matrix DataFrame we started with. The main difference is that, rather than representing a single block matrix, it represents multiple block matrices, with one such matrix per label (phenotype). The schema of this block matrix DataFrame (`reduced_block_df`) will be as follows:

```
|-- header: string (nullable = true)
|-- size: integer (nullable = true)
|-- values: array (nullable = true)
|    |-- element: double (containsNull = true)
|-- header_block: string (nullable = true)
|-- sample_block: string (nullable = true)
|-- sort_key: integer (nullable = true)
|-- mu: double (nullable = true)
|-- sig: double (nullable = true)
|-- alpha: string (nullable = true)
|-- label: string (nullable = true)
```

This schema is the same as the schema of the DataFrame we started with (`block_df`) with two additional columns:

- `alpha`: Name of the alpha value used in fitting the model that produced the values in this row
- `label`: The label corresponding to the values in this row. Since the genotype block matrix X_0 is phenotype-agnostic, the rows in `block_df` were not restricted to any label (phenotype), but the level 1 block matrix X_1 represents ridge model predictions for the labels the reducer was fit with, so each row is associated with a specific label.

The headers in the X_1 block matrix are derived from a combination of the source block in X_0, the alpha value used in fitting the ridge model, and the label they were fit with. These headers are assigned to header blocks that correspond to the chromosome of the source block in X_0.

Example

```
reduced_block_df = reduction.transform()
```

Performing fit and transform in a single step

If the block genotype matrix, phenotype DataFrame, sample block mapping, and covariates are constant for both the model fitting and transformation, the `RidgeReducer.fit_transform` function can be used to do fit and transform in a single step

Example

```
reduced_block_df = reduction.fit_transform()
```

4.3.6 Stage 3. Estimate phenotypic predictors

At this stage, the block matrix X_1 is used to fit a final predictive model that can generate phenotype predictions $\hat{y}$ using

- **For quantitative phenotypes:** the `RidgeRegression` class.
- **For binary phenotypes:** the `LogisticRegression` class.

1. Initialization

- **For quantitative phenotypes:** As with the `RidgeReducer` class, the `RidgeRegression` class is initialized with a list of alpha values. If alpha values are not provided, they will be generated during `RidgeRegression.fit` based on the unique number of headers in the blocked matrix X_1, denoted by h_1, and a set of heritability values.

  $$\alpha = h_1\big[\frac{1}{0.99}, \frac{1}{0.75}, \frac{1}{0.50}, \frac{1}{0.25}, \frac{1}{0.01}\big]$$

  These values are only sensible if the phenotypes are on the scale of one.

  Example

  ```
  regression = RidgeRegression.from_ridge_reduction(reduction)
  ```

- **For binary phenotypes:** Everything is the same except that `LogisticRegression` class is used instead of `RidgeRegression`.

  Example

  ```
  regression = LogisticRidgeRegression.from_ridge_reduction(reduction)
  ```

2. Model fitting

Model fitting is performed using

- **For quantitative phenotypes:** the `RidgeRegression.fit` function.
- **For binary phenotypes:** the `LogisticRegression.fit` function.

This works much in the same way as the `RidgeReducer` *model fitting*, except that it returns an additional DataFrame that reports the cross validation results in optimizing the hyperparameter alpha.

Parameters

- `block_df`: Spark DataFrame representing the reduced block matrix
- `label_df`: Pandas DataFrame containing the target labels used in fitting the ridge models
- `sample_blocks`: Dictionary containing a mapping of sample block IDs to a list of corresponding sample IDs
- `covariate_df`: Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional)

Return

The first output is a model DataFrame analogous to the *model DataFrame* provided by the `RidgeReducer`. An important difference is that the header block ID for all rows will be 'all', indicating that all headers from all blocks have been used in a single fit, rather than fitting within blocks.

The second output is a cross validation report DataFrame containing the results of the hyperparameter (i.e., alpha) value optimization routine. The fields in this DataFrame are:

- `label`: This is the label corresponding to the cross cv results on the row.
- `alpha`: The name of the optimal alpha value
- `r2_mean`: The mean out of fold r2 score for the optimal alpha value

Example

Assuming `regression` is initialized to `RidgeRegression` (for quantitative phenotypes) or `LogisticRegression` (for binary phenotypes) as described *above*, fitting will be done as follows:

```
model_df, cv_df = regression.fit()
```

3. Model transformation

After fitting the model, the model DataFrame and cross validation DataFrame are used to apply the model to the block matrix DataFrame to produce predictions ($\hat{y}$) for each label and sample. This is done using

- **For quantitative phenotypes:** the `RidgeRegression.transform` or `RidgeRegression.transform_loco` method.
- **For binary phenotypes:** the `LogisticRegression.transform` or `LogisticRegression.transform_loco` method.

Here, we describe the leave-one-chromosome-out (LOCO) approach. The input and output of the `transform_loco` function in `RidgeRegression` and `LogisticRegression` are as follows:

Parameters

- `block_df`: Spark DataFrame representing the reduced block matrix
- `label_df`: Pandas DataFrame containing the target labels used in the fitting step
- `sample_blocks`: Dictionary containing a mapping of sample block IDs to a list of corresponding sample IDs
- `model_df`: Spark DataFrame produced by the `RidgeRegression.fit` function (for quantitative phenotypes) or `LogisticRegression.fit` function (for binary phenotypes), representing the reducer model
- `cv_df`: Spark DataFrame produced by the `RidgeRegression.fit` function (for quantitative phenotypes) or `LogisticRegression.fit` function (for binary phenotypes), containing the results of the cross validation routine
- `covariate_df`:
  - **For quantitative phenotypes**: Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).
  - **For binary phenotypes**:
    - If `response='linear'`, `covariate_df` should not be provided.

      Tip: This is because in any follow-up GWAS analysis involving penalization, such as Firth logistic regression, only the linear terms containing genotypes will be used as an offset and covariate coefficients will be refit.

    - If `response='sigmoid'`, a Pandas DataFrame containing covariates to be included in every model in the stacking ensemble.
- `response` (**for binary phenotypes only**): String specifying the desired output. It can be `'linear'` (default) to specify the direct output of the linear WGR model (default) or `'sigmoid'` to specify predicted label probabilities.
- `chromosomes`: List of chromosomes for which to generate a prediction (optional). If not provided, the chromosomes will be inferred from the block matrix.

Return

- **For quantitative phenotypes**: Pandas DataFrame shaped like `label_df`, representing the resulting phenotypic predictors $\hat{y}$, indexed by the sample ID and chromosome with each column representing a single phenotype.
- **For binary phenotypes**:
  - If `response='linear'`: Similar to above but the phenotypic predictor captures only the terms containing genotypes (and not covariates)
  - If `response='sigmoid'`: Pandas DataFrame with the same structure as above containing the predicted probabilities.

Example

Assuming `regression` is initialized to `RidgeRegression` (for quantitative phenotypes) or `LogisticRegression` (for binary phenotypes) as described *above*, fitting will be done as follows:

For quantitative phenotypes:

```
y_hat_df = regression.transform_loco()
```

For binary phenotypes:

```
y_hat_df = regression.transform_loco()
```

4.3.7 Proceed to GWAS

GloWGR GWAS functionality can be used to perform genome-wide association study using the phenotypic predictors to correct for polygenic effects.

4.3.8 Troubleshooting

If you encounter limits related to memory allocation in PyArrow, you may need to tune the number of alphas, number of variants per block, and/or the number of sample blocks. The default values for these hyperparameters are tuned for 500,000 variants and 500,000 samples.

The following values must all be lower than 132,152,839:

- `(# alphas) * (# variants / # variants per block) * (# samples / # sample blocks)`
- `(# alphas * # variants / # variants per block)^2`

Two example notebooks are provided below, the first for quantitative phenotypes and the second for binary phenotypes.

GloWGR notebook for quantitative phenotypes

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/tertiary/glowgr.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/tertiary/glowgr.html" id='-8474209651599240433' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

GloWGR notebook for binary phenotypes

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/tertiary/binaryglowgr.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/tertiary/binaryglowgr.html" id='4440281662649284719' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

4.4 GloWGR: Genome-Wide Association Study (GWAS) Regression Tests

Glow contains functions for performing regression analyses used in genome-wide association studies (GWAS). These functions are best used in conjunction with the *GloWGR whole genome regression method*, but also work as standalone analysis tools.

Tip: Glow automatically converts literal one-dimensional and two-dimensional `numpy` `ndarray` s of `double` s to `array<double>` and `spark.ml` `DenseMatrix` respectively.

4.4.1 Linear regression

`linear_regression` performs a linear regression association test optimized for performance in a GWAS setting. You provide a Spark DataFrame containing the genetic data and Pandas DataFrames with the phenotypes, covariates, and optional offsets (typically predicted phenotypes from GloWGR). The function returns a Spark DataFrame with association test results for each (variant, phenotype) pair.

Each worker node in the cluster tests a subset of the total variant dataset. Multiple phenotypes and variants are tested together to take advantage of efficient matrix-matrix linear algebra primitives.

Example

```
import glow
import numpy as np
import pandas as pd
from pyspark.sql import Row
from pyspark.sql.functions import col, lit

# Read in VCF file
variants = spark.read.format('vcf').load(genotypes_vcf)

# genotype_states returns the number of alt alleles for each sample
# mean_substitute replaces any missing genotype states with the mean of the non-missing␣
↪states
genotypes = (glow.transform('split_multiallelics', variants)
  .withColumn('gt', glow.mean_substitute(glow.genotype_states(col('genotypes'))))
  .select('contigName', 'start', 'names', 'gt')
  .cache())

# Read covariates from a CSV file
covariates = pd.read_csv(covariates_csv, index_col=0)

# Read phenotypes from a CSV file
continuous_phenotypes = pd.read_csv(continuous_phenotypes_csv, index_col=0)

# Run linear regression test
lin_reg_df = glow.gwas.linear_regression(genotypes, continuous_phenotypes, covariates,␣
↪values_column='gt')
```

For complete parameter usage information, check out the API reference for *`glow.gwas.linear_regression()`*.

Note: Glow also includes a SQL-based function for performing linear regression. However, this function only processes one phenotype at time, and so performs more slowly than the batch linear regression function documented above. To read more about the SQL-based function, see the docs for *glow.linear_regression_gwas()*.

4.4.2 Logistic regression

`logistic_regression` performs a logistic regression hypothesis test optimized for performance in a GWAS setting.

Example

```
import glow
import numpy as np
import pandas as pd
from pyspark.sql import Row
from pyspark.sql.functions import col, lit

# Read in VCF file
variants = spark.read.format('vcf').load(genotypes_vcf)

# genotype_states returns the number of alt alleles for each sample
# mean_substitute replaces any missing genotype states with the mean of the non-missing␣
↪states
genotypes = (glow.transform('split_multiallelics', variants)
  .withColumn('gt', glow.mean_substitute(glow.genotype_states(col('genotypes'))))
  .select('contigName', 'start', 'names', 'gt')
  .cache())

# Read covariates from a CSV file
covariates = pd.read_csv(covariates_csv, index_col=0)

# Read phenotypes from a CSV file
binary_phenotypes = pd.read_csv(binary_phenotypes_csv, index_col=0)

# Run logistic regression test with approximate Firth correction for p-values below 0.05
log_reg_df = glow.gwas.logistic_regression(
  genotypes,
  binary_phenotypes,
  covariates,
  correction='approx-firth',
  pvalue_threshold=0.05,
  values_column='gt'
)
```

For complete parameter usage information, check out the API reference for *glow.gwas.logistic_regression()*.

Note: Glow also includes a SQL-based function for performing logistic regression. However, this function only processes one phenotype at time, and so performs more slowly than the batch logistic regression function documented above. To read more about the SQL-based function, see the docs for *glow.logistic_regression_gwas()*.

4.4.3 Offset

The linear and logistic regression functions accept GloWGR phenotypic predictions (either global or per chromosome) as an offset.

```
continuous_offsets = pd.read_csv(continuous_offset_csv, index_col=0)
lin_reg_df = glow.gwas.linear_regression(
  genotypes,
  continuous_phenotypes,
  covariates,
  offset_df=continuous_offsets,
  values_column='gt'
)
```

```
binary_offsets = pd.read_csv(binary_offset_csv, index_col=0)
log_reg_df = glow.gwas.logistic_regression(
  genotypes,
  binary_phenotypes,
  covariates,
  offset_df=binary_offsets,
  correction='approx-firth',
  pvalue_threshold=0.05,
  values_column='gt'
)
```

Tip: The `offset` parameter is especially useful in incorporating the results of *GloWGR* with phenotypes in GWAS. Please refer to *GloWGR: Whole Genome Regression* for details and example notebook.

Example notebooks and blog post

GloWGR: GWAS for quantitative traits

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/tertiary/gwas-quantitative.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/tertiary/gwas-quantitative.html" id='-4617460821588519533' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

GloWGR: GWAS for binary traits

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/tertiary/gwas-binary.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/tertiary/gwas-binary.html" id='-4879829827730212973' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

A detailed example and explanation of a GWAS workflow is available here.

GWAS with MLflow instrumentation

<div class='embedded-notebook'> <a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a style='float:right' href="../_static/notebooks/tertiary/gwas.html">Get notebook link</a></p> <div class='embedded-notebook-container'> <div class='loading-spinner'></div> <iframe src="../_static/notebooks/tertiary/gwas.html" id='3853018510629321465' height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe> </div> </div>

CHAPTER

FIVE

TROUBLESHOOTING

- Job is slow or OOMs (throws an `OutOfMemoryError`) while using an aggregate like `collect_list` or `sample_call_summary_stats`
  - Try disabling the ObjectHashAggregate by setting `spark.sql.execution.useObjectHashAggregateExec` to `false`
- Job is slow or OOMs while writing to partitioned table
  - This error can occur when reading from highly compressed files. Try decreasing `spark.files.maxPartitionBytes` to a smaller value like `33554432` (32MB)
- My VCF looks weird after merging VCFs and saving with `bigvcf`
  - When saving to a VCF, the samples in the genotypes array must be in the same order for each row. This ordering is not guaranteed when using `collect_list` to join multiple VCFs. Try sorting the array using `sort_array`.
- Glow's behavior changed after a release
  - See the Glow release notes. If the Glow release involved a Spark version change, see the Spark migration guide.

CHAPTER SIX

BLOG POSTS

6.1 Introducing GloWGR: An industrial-scale, ultra-fast and sensitive method for genetic association studies

Authors: Leland Barnard, Henry Davidge, Karen Feng, Joelle Mbatchou, Boris Boutkov, Kiavash Kianfar, Lukas Habegger, Jonathan Marchini, Jeffrey Reid, Evan Maxwell, Frank Austin Nothaft

June 22, 2020

The industry partnership between Regeneron and Databricks is enabling innovations in genomics data processing and analysis. Today, we announce that we are making a new whole genome regression method available to the open source bioinformatics community as part of Project Glow.

Large cohorts of individuals with paired clinical and genome sequence data enable unprecedented insight into human disease biology. Population studies such as the UK Biobank, Genomics England, or Genome Asia 100k datasets are driving a need for innovation in methods for working with genetic data. These methods include genome wide association studies (GWAS), which enrich our understanding of the genetic architecture of the disease and are used in cutting-edge industrial applications, such as identifying therapeutic targets for drug development. However, these datasets pose novel statistical and engineering challenges. The statistical challenges have been addressed by tools such as SAIGE and Bolt-LMM, but they are difficult to set up and prohibitively slow to run on biobank-scale datasets.

In a typical GWAS, a single phenotype such as cholesterol levels or diabetes diagnosis status is tested for statistical association with millions of genetic variants across the genome. Sophisticated mixed model and whole genome regression-based approaches have been developed to control for relatedness and population structure inherent to large genetic study populations when testing for genetic associations; several methods such as BOLT-LMM, SAIGE, and fastGWA use a technique called whole genome regression to sensitively analyze a single phenotype in biobank-scale projects. However, deeply phenotyped biobank-scale projects can require tens of thousands of separate GWASs to analyze the full spectrum of clinical variables, and current tools are still prohibitively expensive to run at scale. In order to address the challenge of efficiently analyzing such datasets, the Regeneron Genetics Center has just developed a new approach for the whole-genome regression method that enables running GWAS across upwards of hundreds of phenotypes simultaneously. This exciting new tool provides the same superior test power as current state-of-the-art methods at a small fraction of the computational cost.

This new whole genome regression (WGR) approach recasts the whole genome regression problem to an ensemble model of many small, genetic region-specific models. This method is described in a preprint released today, and implemented in the C++ tool regenie. As part of the collaboration between the Regeneron Genetics Center and Databricks on the open source Project Glow, we are excited to announce GloWGR, a lightning-fast and highly scalable distributed implementation of this WGR algorithm, designed from the ground up with Apache Spark and integrated with other Glow functionality. With GloWGR, performing WGR analyses on dozens of phenotypes can be accomplished simultaneously in a matter of minutes, a task that would require hundreds or thousands of hours with existing state-of-the-art tools. Moreover, GloWGR distributes along both the sample and genetic variant matrix dimensions, allowing for linear scaling and a high degree of data and task parallelism. GloWGR plugs seamlessly into any existing GWAS workflow, providing an immediate boost to association detection power at a negligible computational cost.

6.1.1 Achieving High Accuracy and Efficiency with Whole-Genome Regression

This whole genome regression tool has a number of virtues. First, it is more efficient: as implemented in the single node, open-source regenie tool, whole genome regression is orders of magnitude faster than either SAIGE, Bolt-LMM, or fastGWA, while producing equivalent results (Figure 1). Second, it is straightforward to parallelize: in the next section, we describe how we implemented whole genome regression using Apache Spark™ in the open-source Project Glow.

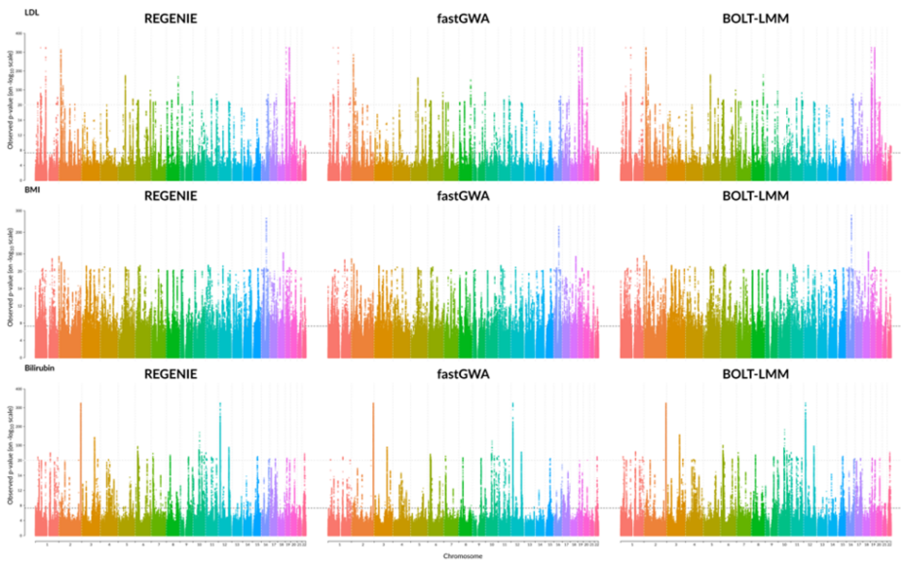


Fig. 6.1: Comparison of GWAS results for three quantitative phenotypes from the UK Biobank project, produced by REGENIE, BOLT-LMM, and fastGWA.

In addition to performance considerations, the whole genome regression approach produces covariates that are compatible with standard GWAS methods, and which eliminate spurious associations caused by population structure that are seen with traditional approaches. The Manhattan plots in figure 2 below compare the results of a traditional linear regression GWAS using standard covariates, to a linear regression GWAS using the covariates generated by WGR. This flexibility of GloWGR is another tremendous advantage over existing GWAS tools, and will allow for a wide variety of exciting extensions to the association testing framework that is already available in Glow.

Figure 3 shows performance comparisons between GloWGR, REGENIE, BoltLMM, and fastGWA. We benchmarked the whole genome regression test implemented in Glow against the C++ implementation available in the single-node regenie tool to validate the accuracy of the method. We found that the two approaches achieve statistically identical results. We also found that the Apache Spark™ based implementation in Glow scales linearly with the number of nodes used.

6.1.2 Scaling Whole Genome Regression within Project Glow

Performing WGR analysis with GloWGR has 5 steps:

- Dividing the genotype matrix into contiguous blocks of SNPs (~1000 SNPs per block, referred to as loci)
- Fitting multiple ridge models (~10) with varying ridge penalties within each locus
- Using the resulting ridge models to reduce the locus from a matrix of 1,000 features to 10 features (each feature is the prediction of one of the ridge models)
- Pooling the resulting features of all loci into a new reduced feature matrix *X* (*N* individuals by *L* loci x *J* ridge models per locus)

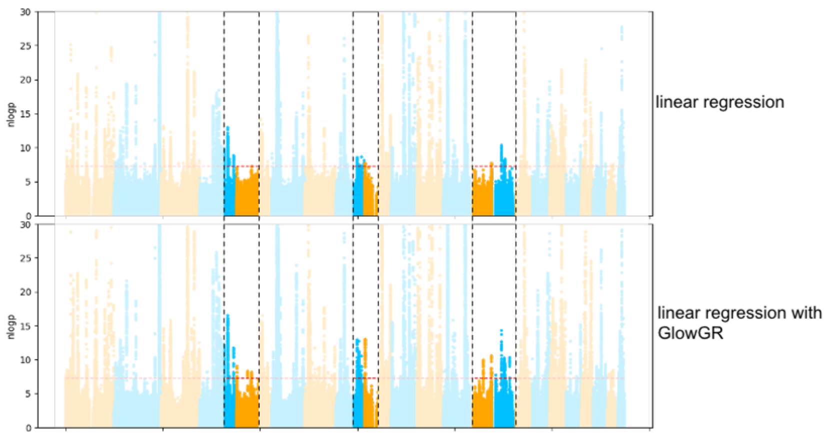


Fig. 6.2: Comparison of GWAS results of the quantitative phenotype bilirubin from the UK Biobank project, evaluated using standard linear regression and linear regression with GloWGR. The heightened peaks in the highlighted regions show the increase in power to detect subtler associations that is gained with GloWGR.

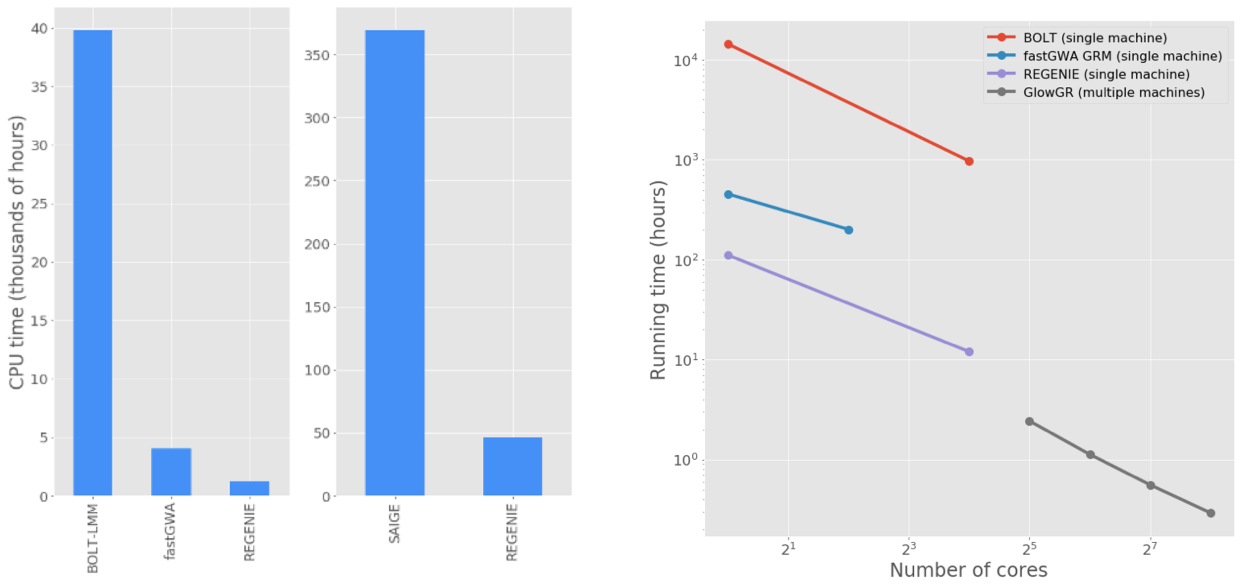


Fig. 6.3: Left: end-to-end GWAS runtime comparison for 50 quantitative traits from the UK Biobank project. Right: Run time comparison to fit WGR models against 50 quantitative phenotypes from the UK Biobank project. GloWGR scales well with cluster size, allowing for modeling of dozens of phenotypes in minutes without costing additional CPU efficiency. The exact list of phenotypes and computation environment details can be found here.

- Fitting a final regularized model from X for the genome-wide contribution to phenotype Y.

Glow provides the easy-to-use abstractions shown in figure 4 for transforming large genotype matrices into the blocked matrix (below, left) and then fitting the whole genome regression model (below, right). These can be applied to data loaded in any of the genotype file formats that Glow understands, including VCF, Plink, and BGEN formats, as well as genotype data stored in Apache Spark™ native file formats like Delta Lake.

```
block_df, sample_blocks = block_variants_and_samples(variant_df,
                                                     sample_ids,
                                                     variants_per_block,
                                                     sample_block_count)
```

```
alphas_regression = np.logspace(1, 4, 10)
estimator = RidgeRegression(alphas_regression)
model_df, cv_df = estimator.fit(reduced_block_df,
                                label_df,
                                sample_blocks,
                                covariates)
```

Fig. 6.4: Creating a matrix grouped by locus and fitting mixed ridge regression models using GloWGR

Glow provides an implementation of the WGR method for quantitative traits, and a binary trait variant is in progress. The covariate-adjusted phenotype created by GloWGR can be written out as an Apache Parquet ™ or Delta Lake dataset, which can easily be loaded by and analyzed within Apache Spark, pandas, and other tools. Ultimately, using the covariates computed with WGR in a genome-wide association study is as simple as running the command shown in Figure 5, below. This command is run by Apache Spark™, in parallel, across all of the genetic markers under test.

```
pdf = (label_df - y_hat_df).reset_index('contigName')
apdf = pdf.melt(id_vars=['contigName']) \
  .groupby(['contigName', 'variable']) \
  .aggregate(lambda x: list(x)) \
  .reset_index() \
  .rename(columns={'variable': 'trait', 'value': 'pt'})
adjusted_phenotypes = spark.createDataFrame(apdf)

wgr_gwas = variant_df.join(adjusted_phenotypes, ['contigName']).select(
'contigName',
'start',
'names',
'trait',
expand_struct(linear_regression_gwas(
  col('values'),
  col('pt'),
  lit(covariates.to_numpy())
)))
```

Fig. 6.5: Updating phenotypes with the WGR results and running a GWAS using the built-in association test methods from Glow

6.1.3 Join us and try whole genome regression in Glow!

Whole genome regression is available in Glow, which is an open source project hosted on Github, with an Apache 2 license. You can get started with this notebook that shows how to use GloWGR on data from 1,000 Genomes, by reading the preprint, by reading our project docs, or you can create a fork of the repository to start contributing code today.

6.2 Glow 0.4 Enables Integration of Genomic Variant and Annotation Data

Author: Kiavash Kianfar
June 9, 2020

Glow 0.4 was released on May 20, 2020. This blog focuses on the highlight of this release, the newly introduced capability to ingest genomic annotation data from the GFF3 (Generic Feature Format Version 3) flat file format. This release also includes other feature and usability improvements, which will be briefly reviewed at the end of this blog.

GFF3 is a sequence annotation flat file format proposed by the Sequence Ontology Project in 2013, which since has become the de facto format for genome annotation and is widely used by genome browsers and databases such as NCBI RefSeq and GenBank. GFF3, a 9-column tab-separated text format, typically carries the majority of the annotation data in the ninth column, called `attributes`, as a semi-colon-separated list of `<tag>=<value>` entries. As a result, although GFF3 files can be read as Spark DataFrames using Spark SQL's standard `csv` data source, the schema of the resulting DataFrame would be quite unwieldy for query and data manipulation of annotation data, because the whole list of attribute tag-value pairs for each sequence will appear as a single semi-colon-separated string in the `attributes` column of the DataFrame.

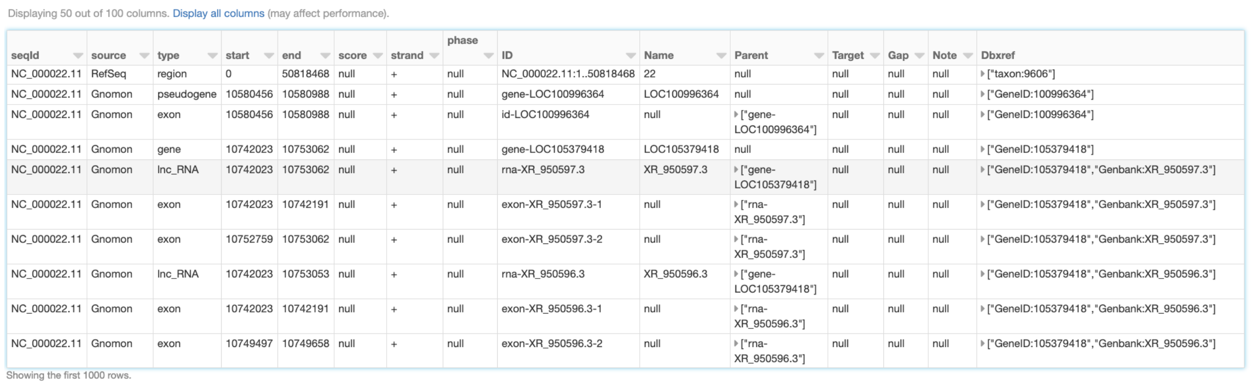
Glow 0.4 adds the new and flexible `gff` Spark SQL data source to address this challenge and create a smooth GFF3 ingest and query experience. While reading the GFF3 file, the `gff` data source parses the `attributes` column of the file to create an appropriately typed column for each tag. In each row, this column will contain the value corresponding to that tag in that row (or `null` if the tag does not appear in the row). Consequently, all tags in the GFF3 `attributes` column will have their own corresponding column in the Spark DataFrame, making annotation data query and manipulation much easier.

6.2.1 Ingesting GFF3 Annotation Data

Like any other Spark data source, reading GFF3 files using Glow's `gff` data source can be done in a single line of code. As an example, we can ingest the annotations of the Homo Sapiens genome assembly GRCh38.p13 from a GFF3 file (obtained from RefSeq ftp site) as shown below. Here, we have also filtered the annotations to chromosome 22 in order to use the resulting `annotations_df` DataFrame (Fig. 6.6) in continuation of our example. The `annotations_df` alias is for the same purpose as well.

```
import glow
glow.register(spark)

gff_path = '/databricks-datasets/genomics/gffs/GCF_000001405.39_GRCh38.p13_genomic.gff.
↪bgz'

annotations_df = spark.read.format('gff').load(gff_path) \
    .filter("seqid = 'NC_000022.11'") \
    .alias('annotations_df')
```

In addition to reading uncompressed `.gff` files, the `gff` data source supports all compression formats supported by Spark's `csv` data source, including `.gz` and `.bgz`. It is strongly recommended to use splittable compression formats like `.bgz` instead of `.gz` for better parallelization of the read process.



Displaying 50 out of 100 columns. Display all columns (may affect performance).

seqId	source	type	start	end	score	strand	phase	ID	Name	Parent	Target	Gap	Note	Dbxref
NC_000022.11	RefSeq	region	0	50818468	null	+	null	NC_000022.11:1..50818468	22	null	null	null	null	["taxon:9606"]
NC_000022.11	Gnomon	pseudogene	10580456	10580988	null	+	null	gene-LOC100996364	LOC100996364	null	null	null	null	["GeneID:100996364"]
NC_000022.11	Gnomon	exon	10580456	10580988	null	+	null	id-LOC100996364	null	["gene-LOC100996364"]	null	null	null	["GeneID:100996364"]
NC_000022.11	Gnomon	gene	10742023	10753062	null	+	null	gene-LOC105379418	LOC105379418	null	null	null	null	["GeneID:105379418"]
NC_000022.11	Gnomon	lnc_RNA	10742023	10753062	null	+	null	rna-XR_950597.3	XR_950597.3	["gene-LOC105379418"]	null	null	null	["GeneID:105379418","Genbank:XR_950597.3"]
NC_000022.11	Gnomon	exon	10742023	10742191	null	+	null	exon-XR_950597.3-1	null	["rna-XR_950597.3"]	null	null	null	["GeneID:105379418","Genbank:XR_950597.3"]
NC_000022.11	Gnomon	exon	10752759	10753062	null	+	null	exon-XR_950597.3-2	null	["rna-XR_950597.3"]	null	null	null	["GeneID:105379418","Genbank:XR_950597.3"]
NC_000022.11	Gnomon	lnc_RNA	10742023	10753053	null	+	null	rna-XR_950596.3	XR_950596.3	["gene-LOC105379418"]	null	null	null	["GeneID:105379418","Genbank:XR_950596.3"]
NC_000022.11	Gnomon	exon	10742023	10742191	null	+	null	exon-XR_950596.3-1	null	["rna-XR_950596.3"]	null	null	null	["GeneID:105379418","Genbank:XR_950596.3"]
NC_000022.11	Gnomon	exon	10749497	10749658	null	+	null	exon-XR_950596.3-2	null	["rna-XR_950596.3"]	null	null	null	["GeneID:105379418","Genbank:XR_950596.3"]

Showing the first 1000 rows.

Fig. 6.6: A small section of the `annotations_df` DataFrame

6.2.2 Schema

Let us have a closer look at the schema of the resulting DataFrame, which was automatically inferred by Glow's `gff` data source:

```
annotations_df.printSchema()
```

```
root
 |-- seqId: string (nullable = true)
 |-- source: string (nullable = true)
 |-- type: string (nullable = true)
 |-- start: long (nullable = true)
 |-- end: long (nullable = true)
 |-- score: double (nullable = true)
 |-- strand: string (nullable = true)
 |-- phase: integer (nullable = true)
 |-- ID: string (nullable = true)
 |-- Name: string (nullable = true)
 |-- Parent: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- Target: string (nullable = true)
 |-- Gap: string (nullable = true)
 |-- Note: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- Dbxref: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- Is_circular: boolean (nullable = true)
 |-- align_id: string (nullable = true)
 |-- allele: string (nullable = true)
 .
 .
 .
 |-- transl_table: string (nullable = true)
 |-- weighted_identity: string (nullable = true)
```

This schema has 100 fields (not all shown here). The first eight fields (`seqId`, `source`, `type`, `start`, `end`, `score`, `strand`, and `phase`), here referred to as the "base" fields, correspond to the first eight columns of the GFF3 format cast in the proper data types. The rest of the fields in the inferred schema are the result of parsing the `attributes` column of the GFF3 file. Fields corresponding to any "official" tag (those referred to as "tags with pre-defined meaning" in the GFF3 format description), if present in the GFF3 file, come first in appropriate data types. The official fields are followed by the "unofficial" fields (fields corresponding to any other tag) in alphabetical order. In the example above, `ID`, `Name`, `Parent`, `Target`, `Gap`, `Note`, `Dbxref`, and `Is_circular` are the official fields, and the rest are the unofficial fields. The `gff` data source discards the comments, directives, and FASTA lines that may be in the GFF3 file.

As it is not uncommon for the official tags to be spelled differently in terms of letter case and underscore usage across different GFF3 files, or even within a single GFF3 file, the `gff` data source is designed to be insensitive to letter case and underscore in extracting official tags from the `attributes` field. For example, the official tag `Dbxref` will be correctly extracted as an official field even if it appears as `dbxref` or `dbx_ref` in the GFF3 file. Please see Glow documentation for more details.

Like other Spark SQL data sources, Glow's `gff` data source is also able to accept a user-specified schema through the `.schema` command. The data source behavior in this case is also designed to be quite flexible. More specifically, the fields (and their types) in the user-specified schema are treated as the list of fields, whether base, official, or unofficial, to be extracted from the GFF3 file (and cast to the specified types). Please see the Glow documentation for more details on how user-specified schemas can be used.

6.2.3 Example: Gene Transcripts and Transcript Exons

With the annotation tags extracted as individual DataFrame columns using Glow's `gff` data source, query and data preparation over genetic annotations becomes as easy as writing common Spark SQL commands in the user's API of choice. As an example, here we demonstrate how simple queries can be used to extract data regarding hierarchical grouping of genomic features from the `annotations_df` created *above*.

One of the main advantages of the GFF3 format compared to older versions of GFF is the improved presentation of feature hierarchies (see GFF3 format description for more details). Two examples of such hierarchies are:

- Transcripts of a gene (here, gene is the "parent" feature and its transcripts are the "children" features).
- Exons of a transcript (here, the transcript is the parent and its exons are the children).

In the GFF3 format, the parents of the feature in each row are identified by the value of the `parent` tag in the `attributes` column, which includes the ID(s) of the parent(s) of the row. Glow's `gff` data source extracts this information as an array of parent ID(s) in a column of the resulting DataFrame called `parent`.

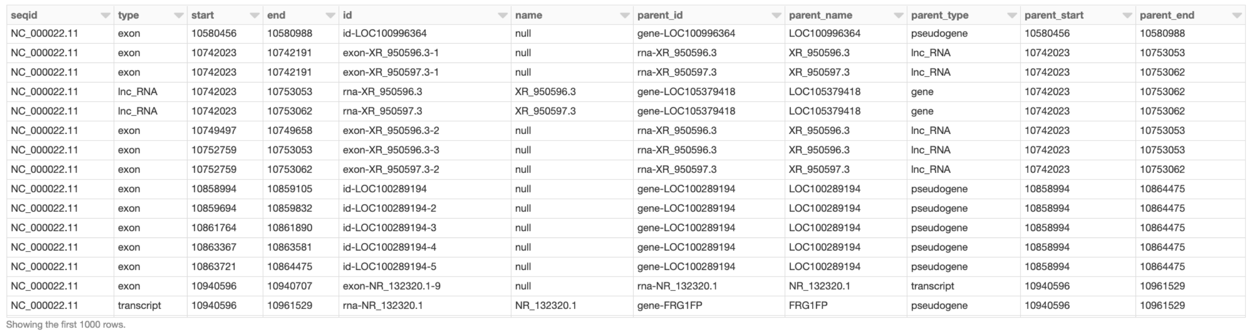
Assume we would like to create a DataFrame, called `gene_transcript_df`, which, for each gene on chromosome 22, provides some basic information about the gene and all its transcripts. As each row in the `annotations_df` of our example has at most a single parent, the `parent_child_df` DataFrame created by the following query will help us in achieving our goal. This query joins `annotations_df` with a subset of its own columns on the `parent` column as the key. Fig. 6.7 shows a small section of `parent_child_df`.

```
from pyspark.sql.functions import *

parent_child_df = annotations_df \
.join(
  annotations_df.select('id', 'type', 'name', 'start', 'end').alias('parent_df'),
  col('annotations_df.parent')[0] == col('parent_df.id') # each row in annotation_df has␣
↪at most one parent
) \
.orderBy('annotations_df.start', 'annotations_df.end') \
.select(
  'annotations_df.seqid',
  'annotations_df.type',
  'annotations_df.start',
  'annotations_df.end',
```

(continues on next page)

(continued from previous page)

```
  'annotations_df.id',
  'annotations_df.name',
  col('annotations_df.parent')[0].alias('parent_id'),
  col('parent_df.Name').alias('parent_name'),
  col('parent_df.type').alias('parent_type'),
  col('parent_df.start').alias('parent_start'),
  col('parent_df.end').alias('parent_end')
) \
.alias('parent_child_df')
```



seqid	type	start	end	id	name	parent_id	parent_name	parent_type	parent_start	parent_end
NC_000022.11	exon	10580456	10580988	id-LOC100996364	null	gene-LOC100996364	LOC100996364	pseudogene	10580456	10580988
NC_000022.11	exon	10742023	10742191	exon-XR_950596.3-1	null	rna-XR_950596.3	XR_950596.3	lnc_RNA	10742023	10753053
NC_000022.11	exon	10742023	10742191	exon-XR_950597.3-1	null	rna-XR_950597.3	XR_950597.3	lnc_RNA	10742023	10753062
NC_000022.11	lnc_RNA	10742023	10753053	rna-XR_950596.3	XR_950596.3	gene-LOC105379418	LOC105379418	gene	10742023	10753062
NC_000022.11	lnc_RNA	10742023	10753062	rna-XR_950597.3	XR_950597.3	gene-LOC105379418	LOC105379418	gene	10742023	10753062
NC_000022.11	exon	10749497	10749658	exon-XR_950596.3-2	null	rna-XR_950596.3	XR_950596.3	lnc_RNA	10742023	10753053
NC_000022.11	exon	10752759	10753053	exon-XR_950596.3-3	null	rna-XR_950596.3	XR_950596.3	lnc_RNA	10742023	10753053
NC_000022.11	exon	10752759	10753062	exon-XR_950597.3-2	null	rna-XR_950597.3	XR_950597.3	lnc_RNA	10742023	10753062
NC_000022.11	exon	10858994	10859105	id-LOC100289194	null	gene-LOC100289194	LOC100289194	pseudogene	10858994	10864475
NC_000022.11	exon	10859694	10859832	id-LOC100289194-2	null	gene-LOC100289194	LOC100289194	pseudogene	10858994	10864475
NC_000022.11	exon	10861764	10861890	id-LOC100289194-3	null	gene-LOC100289194	LOC100289194	pseudogene	10858994	10864475
NC_000022.11	exon	10863367	10863581	id-LOC100289194-4	null	gene-LOC100289194	LOC100289194	pseudogene	10858994	10864475
NC_000022.11	exon	10863721	10864475	id-LOC100289194-5	null	gene-LOC100289194	LOC100289194	pseudogene	10858994	10864475
NC_000022.11	exon	10940596	10940707	exon-NR_132320.1-9	null	rna-NR_132320.1	NR_132320.1	transcript	10940596	10961529
NC_000022.11	transcript	10940596	10961529	rna-NR_132320.1	NR_132320.1	gene-FRG1FP	FRG1FP	pseudogene	10940596	10961529

Showing the first 1000 rows.

Fig. 6.7: A small section of the `parent_child_df` DataFrame

Having the `parent_child_df` DataFrame, we can now write the following simple function, called `parent_child_summary`, which, given this DataFrame, the parent type, and the child type, generates a DataFrame containing basic information on each parent of the given type and all its children of the given type.

```
from pyspark.sql.dataframe import *

def parent_child_summary(parent_child_df: DataFrame, parent_type: str, child_type: str) -
↪> DataFrame:
  return parent_child_df \
    .select(
      'seqid',
      col('parent_id').alias(f'{parent_type}_id'),
      col('parent_name').alias(f'{parent_type}_name'),
      col('parent_start').alias(f'{parent_type}_start'),
      col('parent_end').alias(f'{parent_type}_end'),
      col('id').alias(f'{child_type}_id'),
      col('start').alias(f'{child_type}_start'),
      col('end').alias(f'{child_type}_end'),
    ) \
    .where(f"type == '{child_type}' and parent_type == '{parent_type}'") \
    .groupBy(
      'seqid',
      f'{parent_type}_id',
      f'{parent_type}_name',
      f'{parent_type}_start',
      f'{parent_type}_end'
    ) \
    .agg(
```

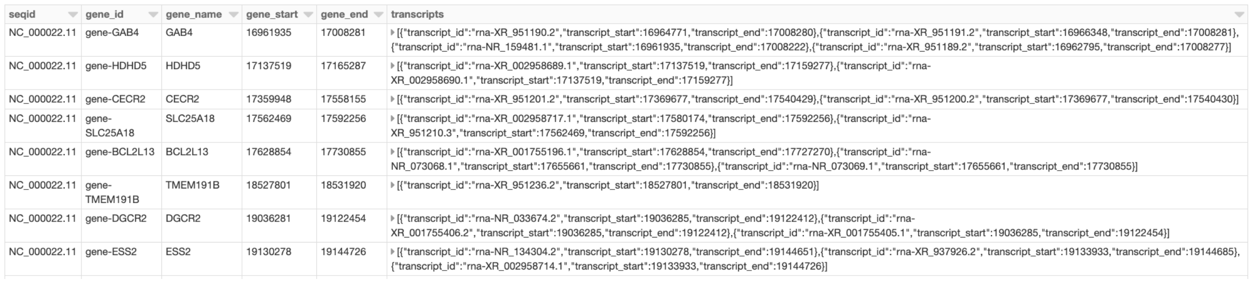
(continues on next page)

(continued from previous page)

```
   collect_list(
    struct(
     f'{child_type}_id',
     f'{child_type}_start',
     f'{child_type}_end'
    )
   ).alias(f'{child_type}s')
  ) \
  .orderBy(
   f'{parent_type}_start',
   f'{parent_type}_end'
  ) \
  .alias(f'{parent_type}_{child_type}_df')
```

Now we can generate our intended `gene_transcript_df` DataFrame, shown in Fig. 6.8, with a single call to this function:

```
gene_transcript_df = parent_child_summary(parent_child_df, 'gene', 'transcript')
```



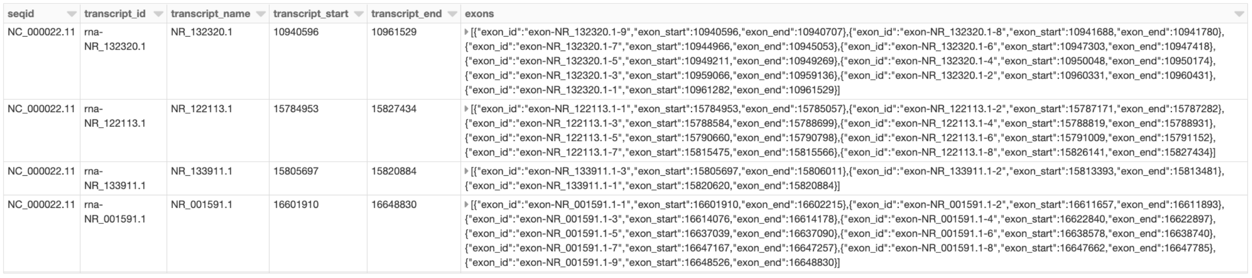
seqid	gene_id	gene_name	gene_start	gene_end	transcripts
NC_000022.11	gene-GAB4	GAB4	16961935	17008281	▸ [{"transcript_id":"rna-XR_951190.2","transcript_start":16964771,"transcript_end":17008280},{"transcript_id":"rna-XR_951191.2","transcript_start":16966348,"transcript_end":17008281}, {"transcript_id":"rna-NR_159481.1","transcript_start":16961935,"transcript_end":17008222},{"transcript_id":"rna-XR_951189.2","transcript_start":16962795,"transcript_end":17008277}]
NC_000022.11	gene-HDHD5	HDHD5	17137519	17165287	▸ [{"transcript_id":"rna-XR_002958689.1","transcript_start":17137519,"transcript_end":17159277},{"transcript_id":"rna-XR_002958690.1","transcript_start":17137519,"transcript_end":17159277}]
NC_000022.11	gene-CECR2	CECR2	17359948	17558155	▸ [{"transcript_id":"rna-XR_951201.2","transcript_start":17369677,"transcript_end":17540429},{"transcript_id":"rna-XR_951200.2","transcript_start":17369677,"transcript_end":17540430}]
NC_000022.11	gene-SLC25A18	SLC25A18	17562469	17592256	▸ [{"transcript_id":"rna-XR_002958717.1","transcript_start":17580174,"transcript_end":17592256},{"transcript_id":"rna-XR_951210.3","transcript_start":17562469,"transcript_end":17592256}]
NC_000022.11	gene-BCL2L13	BCL2L13	17628854	17730855	▸ [{"transcript_id":"rna-XR_001755196.1","transcript_start":17628854,"transcript_end":17727270},{"transcript_id":"rna-NR_073068.1","transcript_start":17655661,"transcript_end":17730855},{"transcript_id":"rna-NR_073069.1","transcript_start":17655661,"transcript_end":17730855}]
NC_000022.11	gene-TMEM191B	TMEM191B	18527801	18531920	▸ [{"transcript_id":"rna-XR_951236.2","transcript_start":18527801,"transcript_end":18531920}]
NC_000022.11	gene-DGCR2	DGCR2	19036281	19122454	▸ [{"transcript_id":"rna-NR_033674.2","transcript_start":19036285,"transcript_end":19122412},{"transcript_id":"rna-XR_001755406.2","transcript_start":19036285,"transcript_end":19122412},{"transcript_id":"rna-XR_001755405.1","transcript_start":19036285,"transcript_end":19122454}]
NC_000022.11	gene-ESS2	ESS2	19130278	19144726	▸ [{"transcript_id":"rna-NR_134304.2","transcript_start":19130278,"transcript_end":19144651},{"transcript_id":"rna-XR_937926.2","transcript_start":19133933,"transcript_end":19144685}, {"transcript_id":"rna-XR_002958714.1","transcript_start":19133933,"transcript_end":19144726}]

Fig. 6.8: A small section of the `gene_transcript_df` DataFrame

In each row of this DataFrame, the `transcripts` column contains the ID, start and end of all transcripts of the gene in that row as an array of structs.

The same function can now be used to generate any parent-child feature summary. For example, we can generate the information of all exons of each transcript on chromosome 22 with another call to the `parent_child_summary` function as shown below. Fig. 6.9 shows the generated `transcript_exon_df` DataFrame.

```
transcript_exon_df = parent_child_summary(parent_child_df, 'transcript', 'exon')
```



seqid	transcript_id	transcript_name	transcript_start	transcript_end	exons
NC_000022.11	rna-NR_132320.1	NR_132320.1	10940596	10961529	▸ [{"exon_id":"exon-NR_132320.1-9","exon_start":10940596,"exon_end":10940707},{"exon_id":"exon-NR_132320.1-8","exon_start":10941688,"exon_end":10941780}, {"exon_id":"exon-NR_132320.1-7","exon_start":10944966,"exon_end":10945053},{"exon_id":"exon-NR_132320.1-6","exon_start":10947303,"exon_end":10947418}, {"exon_id":"exon-NR_132320.1-5","exon_start":10949211,"exon_end":10949269},{"exon_id":"exon-NR_132320.1-4","exon_start":10950048,"exon_end":10950174}, {"exon_id":"exon-NR_132320.1-3","exon_start":10959066,"exon_end":10959136},{"exon_id":"exon-NR_132320.1-2","exon_start":10960331,"exon_end":10960431}, {"exon_id":"exon-NR_132320.1-1","exon_start":10961282,"exon_end":10961529}]
NC_000022.11	rna-NR_122113.1	NR_122113.1	15784953	15827434	▸ [{"exon_id":"exon-NR_122113.1-1","exon_start":15784953,"exon_end":15785057},{"exon_id":"exon-NR_122113.1-2","exon_start":15787171,"exon_end":15787282}, {"exon_id":"exon-NR_122113.1-3","exon_start":15788584,"exon_end":15788699},{"exon_id":"exon-NR_122113.1-4","exon_start":15788819,"exon_end":15788931}, {"exon_id":"exon-NR_122113.1-5","exon_start":15790660,"exon_end":15790798},{"exon_id":"exon-NR_122113.1-6","exon_start":15791009,"exon_end":15791152}, {"exon_id":"exon-NR_122113.1-7","exon_start":15815475,"exon_end":15815566},{"exon_id":"exon-NR_122113.1-8","exon_start":15826141,"exon_end":15827434}]
NC_000022.11	rna-NR_133911.1	NR_133911.1	15805697	15820884	▸ [{"exon_id":"exon-NR_133911.1-3","exon_start":15805697,"exon_end":15806011},{"exon_id":"exon-NR_133911.1-2","exon_start":15813393,"exon_end":15813481}, {"exon_id":"exon-NR_133911.1-1","exon_start":15820620,"exon_end":15820884}]
NC_000022.11	rna-NR_001591.1	NR_001591.1	16601910	16648830	▸ [{"exon_id":"exon-NR_001591.1-1","exon_start":16601910,"exon_end":16602215},{"exon_id":"exon-NR_001591.1-2","exon_start":16611657,"exon_end":16611893}, {"exon_id":"exon-NR_001591.1-3","exon_start":16614076,"exon_end":16614178},{"exon_id":"exon-NR_001591.1-4","exon_start":16622840,"exon_end":16622897}, {"exon_id":"exon-NR_001591.1-5","exon_start":16637039,"exon_end":16637090},{"exon_id":"exon-NR_001591.1-6","exon_start":16638578,"exon_end":16638740}, {"exon_id":"exon-NR_001591.1-7","exon_start":16647167,"exon_end":16647257},{"exon_id":"exon-NR_001591.1-8","exon_start":16647662,"exon_end":16647785}, {"exon_id":"exon-NR_001591.1-9","exon_start":16648526,"exon_end":16648830}]

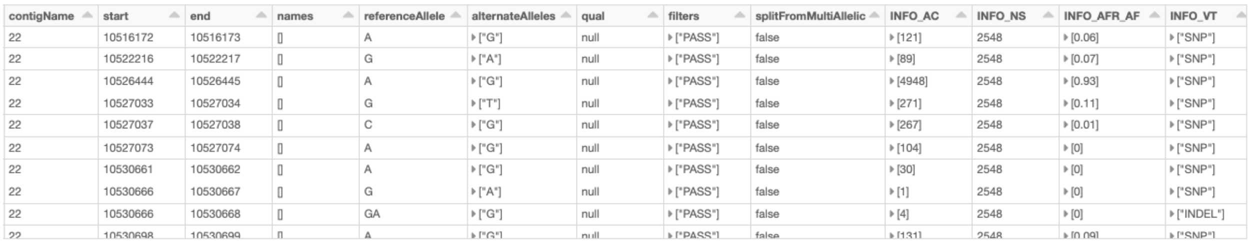
Fig. 6.9: A small section of the `transcript_exon_df` DataFrame

6.2.4 Example Continued: Integration with Variant Data

Glow has *data sources to ingest variant data* from common flat file formats such as VCF, BGEN, and PLINK. Combining the power of Glow's variant data sources with the new `gff` data source, the users can now seamlessly annotate their variant DataFrames by joining them with annotation DataFrames in any desired fashion.

As an example, let us load the chromosome 22 variants of the 1000 Genome Project (on genome assembly GRCh38) from a VCF file (obtained from the project's ftp site). Fig. 6.10 shows the resulting `variants_df`.

```
vcf_path = "/databricks-datasets/genomics/1kg-vcfs/ALL.chr22.shapeit2_integrated_
↪snvindels_v2a_27022019.GRCh38.phased.vcf.gz"

variants_df = spark.read \
  .format("vcf") \
  .load(vcf_path) \
  .alias('variants_df')
```



contigName	start	end	names	referenceAllele	alternateAlleles	qual	filters	splitFromMultiAllelic	INFO_AC	INFO_NS	INFO_AFR_AF	INFO_VT
22	10516172	10516173	[]	A	▸ ["G"]	null	▸ ["PASS"]	false	▸ [121]	2548	▸ [0.06]	▸ ["SNP"]
22	10522216	10522217	[]	G	▸ ["A"]	null	▸ ["PASS"]	false	▸ [89]	2548	▸ [0.07]	▸ ["SNP"]
22	10526444	10526445	[]	A	▸ ["G"]	null	▸ ["PASS"]	false	▸ [4948]	2548	▸ [0.93]	▸ ["SNP"]
22	10527033	10527034	[]	G	▸ ["T"]	null	▸ ["PASS"]	false	▸ [271]	2548	▸ [0.11]	▸ ["SNP"]
22	10527037	10527038	[]	C	▸ ["G"]	null	▸ ["PASS"]	false	▸ [267]	2548	▸ [0.01]	▸ ["SNP"]
22	10527073	10527074	[]	A	▸ ["G"]	null	▸ ["PASS"]	false	▸ [104]	2548	▸ [0]	▸ ["SNP"]
22	10530661	10530662	[]	A	▸ ["G"]	null	▸ ["PASS"]	false	▸ [30]	2548	▸ [0]	▸ ["SNP"]
22	10530666	10530667	[]	G	▸ ["A"]	null	▸ ["PASS"]	false	▸ [1]	2548	▸ [0]	▸ ["SNP"]
22	10530666	10530668	[]	GA	▸ ["G"]	null	▸ ["PASS"]	false	▸ [4]	2548	▸ [0]	▸ ["INDEL"]
22	10530698	10530699	[]	A	▸ ["G"]	null	▸ ["PASS"]	false	▸ [131]	2548	▸ [0.09]	▸ ["SNP"]

Fig. 6.10: A small section of the `variants_df` DataFrame

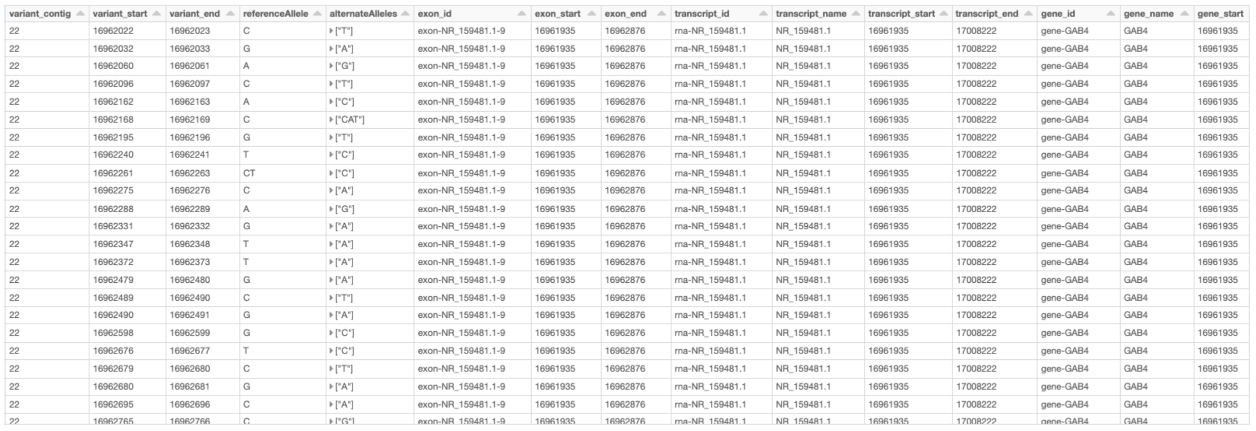
Now using the following double-join query, we can create a DataFrame which, for each variant on a gene on chromosome 22, provides the information of the variant as well as the exon, transcript, and gene on which the variant resides (Fig. 6.11). Note that the first two exploded DataFrames can also be constructed directly from `parent_child_df`. Here, since we had already defined `gene_transcrip_df` and `transcript_exon_df`, we generated these exploded DataFrames simply by applying the `explode` function followed by Glow's *expand_struct* function on them.

```
from glow.functions import *

gene_transcript_exploded_df = gene_transcript_df \
  .withColumn('transcripts', explode('transcripts')) \
  .withColumn('transcripts', expand_struct('transcripts')) \
  .alias('gene_transcript_exploded_df')

transcript_exon_exploded_df = transcript_exon_df \
  .withColumn('exons', explode('exons')) \
  .withColumn('exons', expand_struct('exons')) \
  .alias('transcript_exon_exploded_df')

variant_exon_transcript_gene_df = variants_df \
.join(
  transcript_exon_exploded_df,
  (variants_df.start < transcript_exon_exploded_df.exon_end) &
  (transcript_exon_exploded_df.exon_start < variants_df.end)
) \
.join(
  gene_transcript_exploded_df,
```

(continues on next page)

(continued from previous page)

```
  transcript_exon_exploded_df.transcript_id == gene_transcript_exploded_df.transcript_id
) \
.select(
  col('variants_df.contigName').alias('variant_contig'),
  col('variants_df.start').alias('variant_start'),
  col('variants_df.end').alias('variant_end'),
  col('variants_df.referenceAllele'),
  col('variants_df.alternateAlleles'),
  'transcript_exon_exploded_df.exon_id',
  'transcript_exon_exploded_df.exon_start',
  'transcript_exon_exploded_df.exon_end',
  'transcript_exon_exploded_df.transcript_id',
  'transcript_exon_exploded_df.transcript_name',
  'transcript_exon_exploded_df.transcript_start',
  'transcript_exon_exploded_df.transcript_end',
  'gene_transcript_exploded_df.gene_id',
  'gene_transcript_exploded_df.gene_name',
  'gene_transcript_exploded_df.gene_start',
  'gene_transcript_exploded_df.gene_end'
) \
.orderBy(
  'variant_contig',
  'variant_start',
  'variant_end'
)
```



variant_contig	variant_start	variant_end	referenceAllele	alternateAlleles	exon_id	exon_start	exon_end	transcript_id	transcript_name	transcript_start	transcript_end	gene_id	gene_name	gene_start
22	16962022	16962023	C	["T"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962032	16962033	G	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962060	16962061	A	["G"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962096	16962097	C	["T"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962162	16962163	A	["C"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962168	16962169	C	["CAT"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962195	16962196	G	["T"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962240	16962241	T	["C"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962261	16962263	CT	["C"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962275	16962276	C	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962288	16962289	A	["G"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962331	16962332	G	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962347	16962348	T	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962372	16962373	T	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962479	16962480	G	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962489	16962490	C	["T"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962490	16962491	G	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962598	16962599	G	["C"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962676	16962677	T	["C"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962679	16962680	C	["T"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962680	16962681	G	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962695	16962696	C	["A"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935
22	16962765	16962766	C	["G"]	exon-NR_159481.1-9	16961935	16962876	rna-NR_159481.1	NR_159481.1	16961935	17008222	gene-GAB4	GAB4	16961935

Fig. 6.11: A small section of the `variant_exon_transcript_gene_df` DataFrame

6.2.5 Other Features and Improvements

In addition to the new `gff` reader, Glow 0.4 introduced other features and improvements. A new function, called `mean_substitute`, was introduced, which can be used to substitute the missing values of a numeric Spark array with the mean of the non-missing values. The `normalize_variants` transformer now accepts reference genomes in bgzipped fasta format in addition to the uncompressed fasta. The VCF reader was updated to be able to handle reading file globs that include tabix index files. In addition, this reader no longer has the `splitToBiallelic` option. The `split_multiallelics` transformer introduced in Glow 0.3 can be used instead. Also, the `pipe` transformer was improved so that it does not pipe empty partitions. As a result, users do not need to `repartition` or `coalesce` when piping VCF files. For a complete list of new features and improvements in Glow 0.4, please refer to Glow 0.4 Release Notes.

6.2.6 Try It!

Try Glow 0.4 and its new features here.

6.3 Glow 0.3.0 Introduces Several New Large-Scale Genomic Analysis Features

Author: Kiavash Kianfar
March 2, 2020

Glow 0.3.0 was released on February 21, 2020, improving Glow's power and ease of use in performing large-scale genomic analysis. In this blog, we highlight features and improvements introduced in this release.

6.3.1 Python and Scala APIs for Glow SQL functions

In this release, Python and Scala APIs were introduced for all Glow SQL functions, similar to what is available for Spark SQL functions. In addition to improved simplicity, this provides enhanced compile-time safety. The SQL functions and their Python and Scala clients are generated from the same source so any new functionality in the future will always appear in all three languages. Please refer to *PySpark Functions* for more information on Python APIs for these functions. As an example, the usage of such Python and Scala APIs for the function `normalize_variant` is presented at *the end of next section*.

6.3.2 Improved variant normalization

The variant normalizer received a major improvement in this release. It still behaves like bcftools norm and vt normalize, but is about 2.5x faster and has a more flexible API. Moreover, the new normalizer is implemented as a function in addition to a transformer.

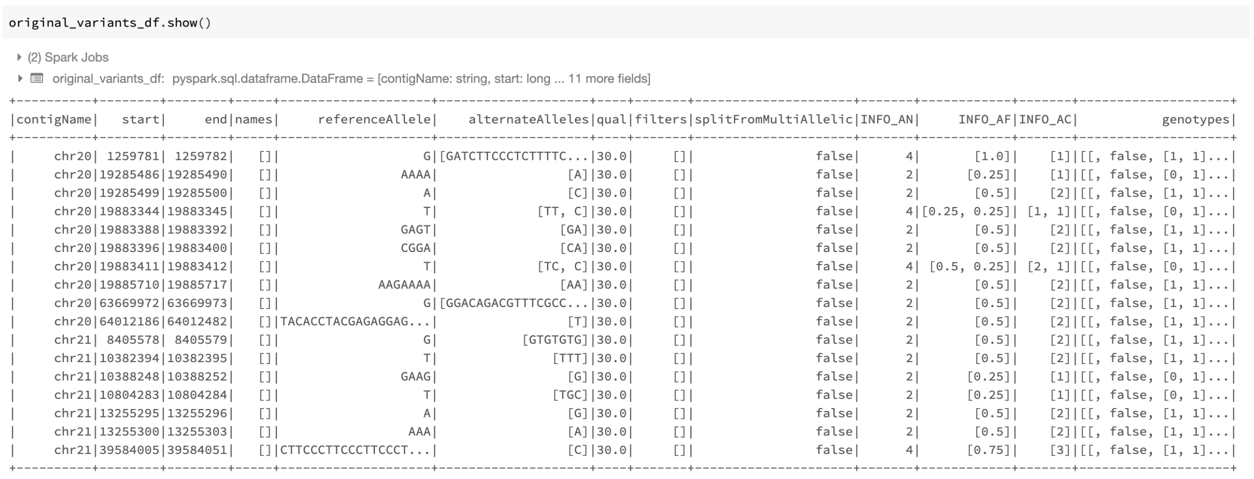
`normalize_variants` **transformer**: The improved transformer preserves the columns of the input DataFrame, adds the normalization status to the DataFrame, and has the option of adding the normalization results (including the normalized coordinates and alleles) to the DataFrame as a new column. As an example, assume we read the `original_variants_df` DataFrame shown in Fig. 6.12 by issuing the following command:

```
original_variants_df = spark.read \
  .format("vcf") \
```

(continues on next page)

(continued from previous page)

```
  .option("includeSampleIds", False) \
  .load("/databricks-datasets/genomics/call-sets")
```



```
original_variants_df.show()
```

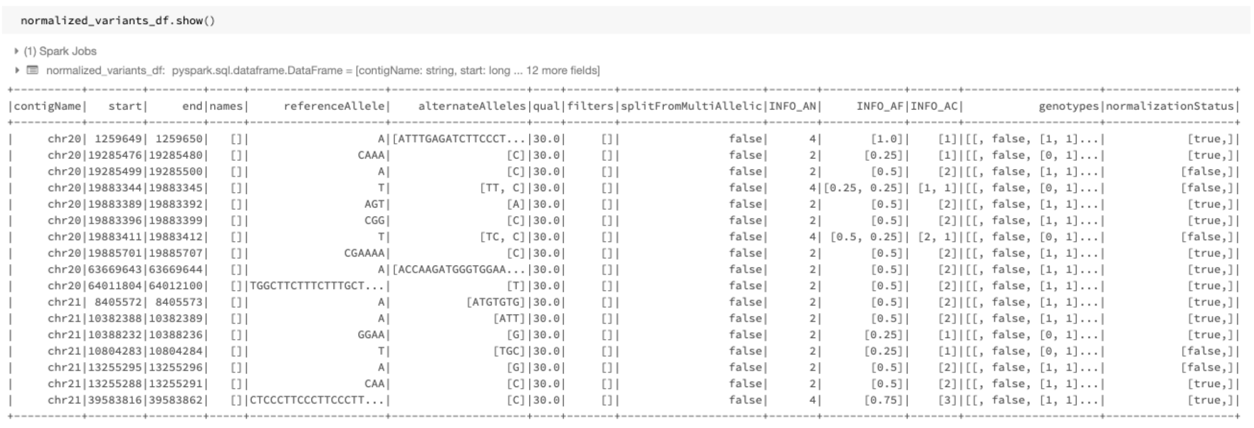
▸ (2) Spark Jobs
▸ original_variants_df: pyspark.sql.dataframe.DataFrame = [contigName: string, start: long ... 11 more fields]

```
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
|contigName|   start|     end|names|     referenceAllele|    alternateAlleles|qual|filters|splitFromMultiAllelic|INFO_AN|     INFO_AF|INFO_AC|           genotypes|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
|     chr20| 1259781| 1259782|   []|                   G|[GATCTTCCCTCTTTTC...|30.0|     []|                false|      4|       [1.0]|    [1]|[[, false, [1, 1]...|
|     chr20|19285486|19285490|   []|                AAAA|                 [A]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr20|19285499|19285500|   []|                   A|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883344|19883345|   []|                   T|             [TT, C]|30.0|     []|                false|      4|[0.25, 0.25]| [1, 1]|[[, false, [0, 1]...|
|     chr20|19883388|19883392|   []|                GAGT|                [GA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883396|19883400|   []|                CGGA|                [CA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883411|19883412|   []|                   T|             [TC, C]|30.0|     []|                false|      4| [0.5, 0.25]| [2, 1]|[[, false, [0, 1]...|
|     chr20|19885710|19885717|   []|             AAGAAAA|                [AA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|63669972|63669973|   []|                   G|[GGACAGACGTTTCGCC...|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|64012186|64012482|   []|TACACCTACGAGAGGAG...|                 [T]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21| 8405578| 8405579|   []|                   G|           [GTGTGTG]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|10382394|10382395|   []|                   T|               [TTT]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|10388248|10388252|   []|                GAAG|                 [G]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr21|10804283|10804284|   []|                   T|               [TGC]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr21|13255295|13255296|   []|                   A|                 [G]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|13255300|13255303|   []|                 AAA|                 [A]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|39584005|39584051|   []|CTTCCCTTCCCTTCCCT...|                 [C]|30.0|     []|                false|      4|      [0.75]|    [3]|[[, false, [1, 1]...|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
```

Fig. 6.12: The variant DataFrame `original_variants_df`

The improved normalizer transformer can be applied on this DataFrame using the following command similar to the previous version of the normalizer:

```
import glow
normalized_variants_df = glow.transform("normalize_variants", \
  original_variants_df, \
  reference_genome_path="/mnt/dbnucleus/dbgenomics/grch38/data/GRCh38_full_analysis_set_
↪plus_decoy_hla.fa" \
)
```



```
normalized_variants_df.show()
```

▸ (1) Spark Jobs
▸ normalized_variants_df: pyspark.sql.dataframe.DataFrame = [contigName: string, start: long ... 12 more fields]

```
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+-------------------+
|contigName|   start|     end|names|     referenceAllele|    alternateAlleles|qual|filters|splitFromMultiAllelic|INFO_AN|     INFO_AF|INFO_AC|           genotypes|normalizationStatus|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+-------------------+
|     chr20| 1259649| 1259650|   []|                   A|[ATTTGAGATCTTCCCT...|30.0|     []|                false|      4|       [1.0]|    [1]|[[, false, [1, 1]...|            [true,]|
|     chr20|19285476|19285480|   []|                CAAA|                 [C]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|            [true,]|
|     chr20|19285499|19285500|   []|                   A|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|           [false,]|
|     chr20|19883344|19883345|   []|                   T|             [TT, C]|30.0|     []|                false|      4|[0.25, 0.25]| [1, 1]|[[, false, [0, 1]...|           [false,]|
|     chr20|19883389|19883392|   []|                 AGT|                 [A]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|            [true,]|
|     chr20|19883396|19883399|   []|                 CGG|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|            [true,]|
|     chr20|19883411|19883412|   []|                   T|             [TC, C]|30.0|     []|                false|      4| [0.5, 0.25]| [2, 1]|[[, false, [0, 1]...|           [false,]|
|     chr20|19885701|19885707|   []|              CGAAAA|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|            [true,]|
|     chr20|63669643|63669644|   []|                   A|[ACCAAGATGGGTGGAA...|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|            [true,]|
|     chr20|64011804|64012100|   []|TGGCTTCTTTCTTTGCT...|                 [T]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|            [true,]|
|     chr21| 8405572| 8405573|   []|                   A|           [ATGTGTG]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|            [true,]|
|     chr21|10382388|10382389|   []|                   A|               [ATT]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|            [true,]|
|     chr21|10388232|10388236|   []|                GGAA|                 [G]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|            [true,]|
|     chr21|10804283|10804284|   []|                   T|               [TGC]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|           [false,]|
|     chr21|13255295|13255296|   []|                   A|                 [G]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|           [false,]|
|     chr21|13255288|13255291|   []|                 CAA|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|            [true,]|
|     chr21|39583816|39583862|   []|CTCCCTTCCCTTCCCTT...|                 [C]|30.0|     []|                false|      4|      [0.75]|    [3]|[[, false, [1, 1]...|            [true,]|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+-------------------+
```

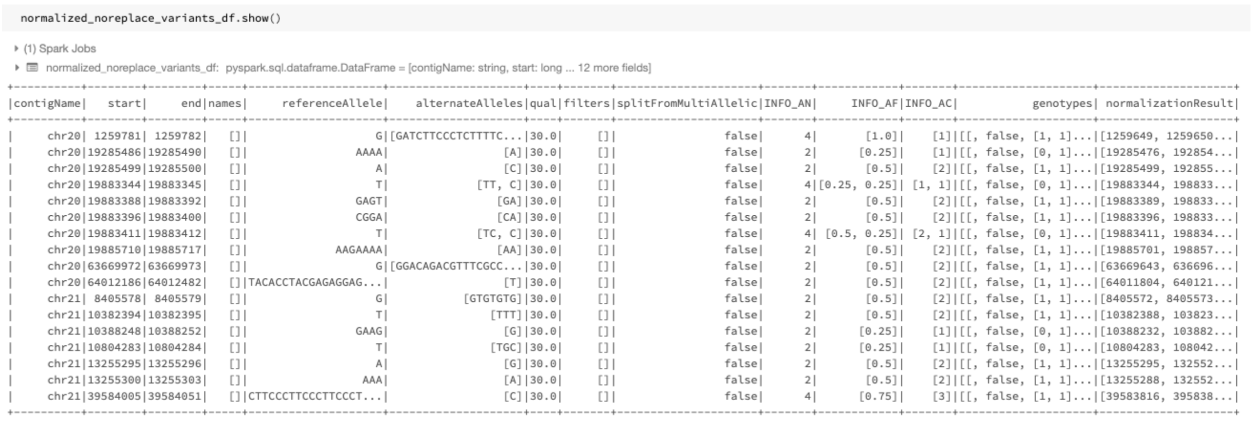
Fig. 6.13: The normalized DataFrame `normalized_variants_df`

The output DataFrame of this improved transformer looks like Fig. 6.13. The `start`, `end`, `referenceAllele`, and `alternateAlleles` fields are updated with the normalized values and a `normalizationStatus` column is added to the DataFrame. This column contains a `changed` subfield indicating whether normalization changed the variant and an `errorMessage` subfield containing the error message in case of an error.

The newly introduced `replace_columns` option can be used to add the normalization results as a new column to the

DataFrame instead of replacing the original `start`, `end`, `referenceAllele`, and `alternateAlleles` fields. This can be done as follows:

```python
import glow
normalized_variants_df = glow.transform("normalize_variants",\
  original_variants_df, \
  replace_columns="False", \
  reference_genome_path="/mnt/dbnucleus/dbgenomics/grch38/data/GRCh38_full_analysis_set_
↪plus_decoy_hla.fa" \
)
```



```
normalized_noreplace_variants_df.show()
```

▸ (1) Spark Jobs

▸ normalized_noreplace_variants_df: pyspark.sql.dataframe.DataFrame = [contigName: string, start: long ... 12 more fields]

```
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+--------------------+
|contigName|   start|     end|names|     referenceAllele|    alternateAlleles|qual|filters|splitFromMultiAllelic|INFO_AN|     INFO_AF|INFO_AC|           genotypes| normalizationResult|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+--------------------+
|     chr20| 1259781| 1259782|   []|                   G|[GATCTTCCCTCTTTTC...|30.0|     []|                false|      4|       [1.0]|    [1]|[[, false, [1, 1]...|[1259649, 1259650...|
|     chr20|19285486|19285490|   []|                AAAA|                 [A]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|[19285476, 192854...|
|     chr20|19285499|19285500|   []|                   A|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[19285499, 192855...|
|     chr20|19883344|19883345|   []|                   T|             [TT, C]|30.0|     []|                false|      4|[0.25, 0.25]| [1, 1]|[[, false, [0, 1]...|[19883344, 198833...|
|     chr20|19883388|19883392|   []|                GAGT|                [GA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[19883389, 198833...|
|     chr20|19883396|19883400|   []|                CGGA|                [CA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[19883396, 198833...|
|     chr20|19883411|19883412|   []|                   T|             [TC, C]|30.0|     []|                false|      4| [0.5, 0.25]| [2, 1]|[[, false, [0, 1]...|[19883411, 198834...|
|     chr20|19885710|19885717|   []|             AAGAAAA|                [AA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[19885701, 198857...|
|     chr20|63669972|63669973|   []|                   G|[GGACAGACGTTTCGCC...|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[63669643, 636696...|
|     chr20|64012186|64012482|   []|TACACCTACGAGAGGAG...|                 [T]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[64011804, 640121...|
|     chr21| 8405578| 8405579|   []|                   G|           [GTGTGTG]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[8405572, 8405573...|
|     chr21|10382394|10382395|   []|                   T|               [TTT]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[10382388, 103823...|
|     chr21|10388248|10388252|   []|                GAAG|                 [G]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|[10388232, 103882...|
|     chr21|10804283|10804284|   []|                   T|               [TGC]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|[10804283, 108042...|
|     chr21|13255295|13255296|   []|                   A|                 [G]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[13255295, 132552...|
|     chr21|13255300|13255303|   []|                 AAA|                 [A]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|[13255288, 132552...|
|     chr21|39584005|39584051|   []|CTTCCCTTCCCTTCCCT...|                 [C]|30.0|     []|                false|      4|      [0.75]|    [3]|[[, false, [1, 1]...|[39583816, 395838...|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+--------------------+
```

Fig. 6.14: The normalized DataFrame `normalized_noreplace_variants_df` with normalization results added as a new column

The resulting DataFrame will be as shown in Fig. 6.14, where a `normalizationResults` column containing the normalized `start`, `end`, `referenceAllele`, `alternateAlleles`, and `normalizationStatus` subfields is added to the DataFrame.

We also note that since the multiallelic variant splitter is implemented as a separate transformer in this release (see below), the `mode` option of the `normalize_variants` transformer is deprecated. Refer to *Variant Normalization* for more details on the `normalize_variants` transformer.

`normalize_variant` function: As mentioned *above*, in this release, variant normalization can also be performed using the newly introduced `normalize_variant` SQL expression function as shown below:

```python
from pyspark.sql.functions import expr
function_normalized_variants_df = original_variants_df.withColumn( \
  "normalizationResult", \
  expr("normalize_variant(contigName, start, end, referenceAllele, alternateAlleles, '/
↪mnt/dbnucleus/dbgenomics/grch38/data/GRCh38_full_analysis_set_plus_decoy_hla.fa')") \
)
```

As discussed in the previous *section*, this SQL expression function, like any other in Glow, now has Python and Scala APIs as well. Therefore, the same can be done in Python as follows:

```python
from glow.functions import normalize_variant
function_normalized_variants_df = original_variants_df.withColumn( \
  "normalizationResult", \
  normalize_variant( \
    "contigName", \
```

(continues on next page)

(continued from previous page)

```
    "start", \
    "end", \
    "referenceAllele", \
    "alternateAlleles", \
    "/mnt/dbnucleus/dbgenomics/grch38/data/GRCh38_full_analysis_set_plus_decoy_hla.fa" \
  ) \
)
```

and in Scala as well, assuming `original_variant_df` is defined in Scala:

```scala
import io.projectglow.functions.normalize_variant
import org.apache.spark.sql.functions.col
val function_normalized_variants_df = original_variants_df.withColumn(
  "normalizationResult",
  normalize_variant(
    col("contigName"),
    col("start"),
    col("end"),
    col("referenceAllele"),
    col("alternateAlleles"),
    "/mnt/dbnucleus/dbgenomics/grch38/data/GRCh38_full_analysis_set_plus_decoy_hla.fa"
  )
)
```

The result of any of the above commands will be the same as Fig. 6.14.

6.3.3 A new transformer to split multiallelic variants

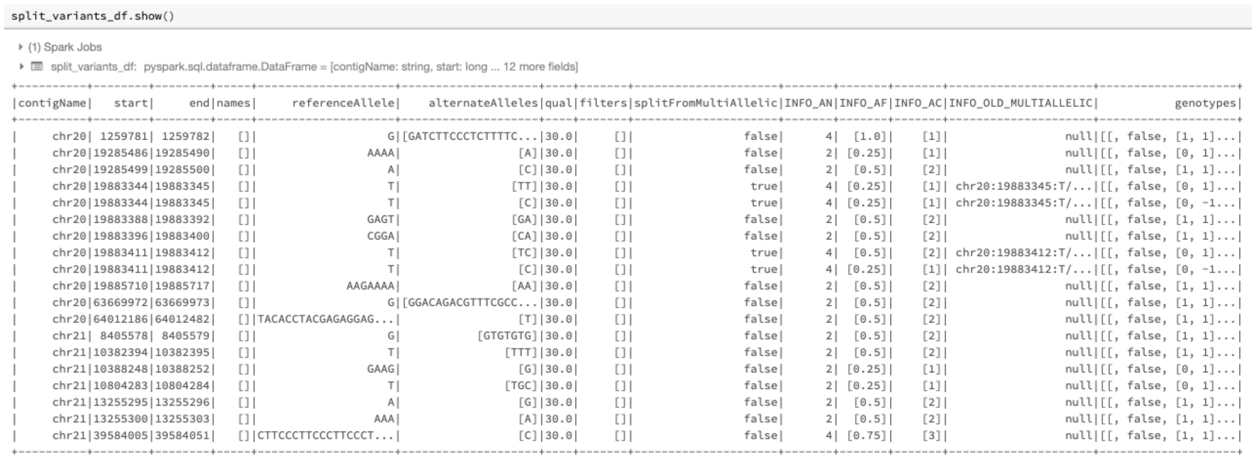
This release also introduced a new DataFrame transformer, called `split_multiallelics`, to split multiallelic variants into biallelic ones with a behavior similar to vt decompose with `-s` option. This behavior is significantly more powerful than the behavior of the previous version's splitter which behaved like GATK's LeftAlignAndTrimVariants with `--split-multi-allelics`. In particular, the array-type `INFO` and genotype fields with elements corresponding to reference and alternate alleles are split "smart"ly (see `-s` option of vt decompose) into biallelic rows. So are the array-type genotype fields with elements sorted in colex order of genotype calles, e.g., the `GL`, `PL`, and `GP` fields in the VCF format. Moreover, an `OLD_MULTIALLELIC` `INFO` field is added to the DataFrame to store the original multiallelic form of the split variants.

The following is an example of using the `split_multiallelic` transformer on the `original_variants_df`. The resulting DataFrame is as in Fig. 6.15.

```python
import glow
split_variants_df = glow.transform("split_multiallelics", original_variants_df)
```

Please note that the new splitter is implemented as a separate transformer from the `normalize_variants` transformer. Previously, splitting could only be done as one of the operation modes of the `normalize_variants` transformer using the now-deprecated mode option.

Please refer to the *documentation of the split_multiallelics transformer* for complete details on the bahavior of this new transformer.



```
split_variants_df.show()
```

▸ (1) Spark Jobs
▸ split_variants_df: pyspark.sql.dataframe.DataFrame = [contigName: string, start: long ... 12 more fields]

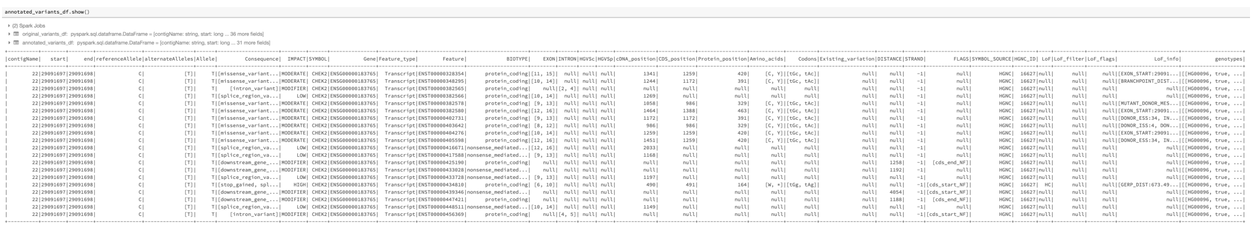
```
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+-------+-------+--------------------+--------------------+
|contigName|   start|     end|names|     referenceAllele|    alternateAlleles|qual|filters|splitFromMultiAllelic|INFO_AN|INFO_AF|INFO_AC|INFO_OLD_MULTIALLELIC|           genotypes|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+-------+-------+--------------------+--------------------+
|     chr20| 1259781| 1259782|   []|                   G|[GATCTTCCCTCTTTTC...|30.0|     []|                false|      4|  [1.0]|    [1]|                null|[[, false, [1, 1]...|
|     chr20|19285486|19285490|   []|                AAAA|                 [A]|30.0|     []|                false|      2| [0.25]|    [1]|                null|[[, false, [0, 1]...|
|     chr20|19285499|19285500|   []|                   A|                 [C]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr20|19883344|19883345|   []|                   T|                [TT]|30.0|     []|                 true|      4| [0.25]|    [1]| chr20:19883345:T/...|[[, false, [0, 1]...|
|     chr20|19883344|19883345|   []|                   T|                 [C]|30.0|     []|                 true|      4| [0.25]|    [1]| chr20:19883345:T/...|[[, false, [0, -1...|
|     chr20|19883388|19883392|   []|                GAGT|                [GA]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr20|19883396|19883400|   []|                CGGA|                [CA]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr20|19883411|19883412|   []|                   T|                [TC]|30.0|     []|                 true|      4|  [0.5]|    [2]| chr20:19883412:T/...|[[, false, [0, 1]...|
|     chr20|19883411|19883412|   []|                   T|                 [C]|30.0|     []|                 true|      4| [0.25]|    [1]| chr20:19883412:T/...|[[, false, [0, -1...|
|     chr20|19885710|19885717|   []|             AAGAAAA|                [AA]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr20|63669972|63669973|   []|                   G|[GGACAGACGTTTCGCC...|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr20|64012186|64012482|   []|TACACCTACGAGAGGAG...|                 [T]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr21| 8405578| 8405579|   []|                   G|           [GTGTGTG]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr21|10382394|10382395|   []|                   T|               [TTT]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr21|10388248|10388252|   []|                GAAG|                 [G]|30.0|     []|                false|      2| [0.25]|    [1]|                null|[[, false, [0, 1]...|
|     chr21|10804283|10804284|   []|                   T|               [TGC]|30.0|     []|                false|      2| [0.25]|    [1]|                null|[[, false, [0, 1]...|
|     chr21|13255295|13255296|   []|                   A|                 [G]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr21|13255300|13255303|   []|                 AAA|                 [A]|30.0|     []|                false|      2|  [0.5]|    [2]|                null|[[, false, [1, 1]...|
|     chr21|39584005|39584051|   []|CTTCCCTTCCCTTCCCT...|                 [C]|30.0|     []|                false|      4| [0.75]|    [3]|                null|[[, false, [1, 1]...|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+-------+-------+--------------------+--------------------+
```

Fig. 6.15: The split DataFrame `split_variants_df`

6.3.4 Parsing of Annotation Fields

The VCF reader and pipe transformer now parse variant annotations from tools such as SnpEff and VEP. This flattens the ANN and CSQ INFO fields, simplifying and accelerating queries on annotations. See the following query and its result in Fig. 6.16 for an example.

```python
from pyspark.sql.functions import expr
variants_df = spark.read\
  .format("vcf")\
  .load("dbfs:/databricks-datasets/genomics/vcfs/loftee.vcf")
annotated_variants_df = original_variants_df.withColumn( \
  "Exploded_INFO_CSQ", \
  expr("explode(INFO_CSQ)") \
) \
.selectExpr("contigName", \
   "start", \
   "end", \
   "referenceAllele", \
   "alternateAlleles", \
   "expand_struct(Exploded_INFO_CSQ)", \
   "genotypes" \
)
```



```
annotated_variants_df.show()
```

▸ (2) Spark Jobs
▸ original_variants_df: pyspark.sql.dataframe.DataFrame = [contigName: string, start: long ... 36 more fields]
▸ annotated_variants_df: pyspark.sql.dataframe.DataFrame = [contigName: string, start: long ... 31 more fields]

```
|contigName|   start|     end|referenceAllele|alternateAlleles|Allele|         Consequence|  IMPACT|SYMBOL|           Gene|Feature_type|        Feature|             BIOTYPE|    EXON|INTRON|HGVSc|HGVSp|cDNA_position|CDS_position|Protein_position|Amino_acids|    Codons|Existing_variation|DISTANCE|STRAND|         FLAGS|SYMBOL_SOURCE|HGNC_ID| LoF|LoF_filter|LoF_flags|            LoF_info|           genotypes|
|        22|29091697|29091698|              C|             [T]|     T|[missense_variant...|MODERATE| CHEK2|ENSG00000183765|  Transcript|ENST00000328354|      protein_coding|[11, 15]|  null| null| null|         1341|        1259|             420|     [C, Y]|[tGc, tAc]|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|[EXON_START:29091...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[missense_variant...|MODERATE| CHEK2|ENSG00000183765|  Transcript|ENST00000348295|      protein_coding|[10, 14]|  null| null| null|         1244|        1172|             391|     [C, Y]|[tGc, tAc]|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|[BRANCHPOINT_DIST...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|    [intron_variant]|MODIFIER| CHEK2|ENSG00000183765|  Transcript|ENST00000382565|      protein_coding|    null|[2, 4]| null| null|         null|        null|            null|       null|      null|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[splice_region_va...|     LOW| CHEK2|ENSG00000183765|  Transcript|ENST00000382566|      protein_coding|[10, 14]|  null| null| null|         1269|        null|            null|       null|      null|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[missense_variant...|MODERATE| CHEK2|ENSG00000183765|  Transcript|ENST00000382578|      protein_coding| [9, 13]|  null| null| null|         1058|         986|             329|     [C, Y]|[tGc, tAc]|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|[MUTANT_DONOR_MES...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[missense_variant...|MODERATE| CHEK2|ENSG00000183765|  Transcript|ENST00000382580|      protein_coding|[12, 16]|  null| null| null|         1464|        1388|             463|     [C, Y]|[tGc, tAc]|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|[EXON_START:29091...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[missense_variant...|MODERATE| CHEK2|ENSG00000183765|  Transcript|ENST00000402731|      protein_coding| [9, 13]|  null| null| null|         1172|        1172|             391|     [C, Y]|[tGc, tAc]|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|[DONOR_ESS:34, IN...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[missense_variant...|MODERATE| CHEK2|ENSG00000183765|  Transcript|ENST00000403642|      protein_coding| [8, 12]|  null| null| null|          986|         986|             329|     [C, Y]|[tGc, tAc]|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|[DONOR_ISS:4, DON...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[missense_variant...|MODERATE| CHEK2|ENSG00000183765|  Transcript|ENST00000404276|      protein_coding|[10, 14]|  null| null| null|         1259|        1259|             420|     [C, Y]|[tGc, tAc]|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|[EXON_START:29091...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[missense_variant...|MODERATE| CHEK2|ENSG00000183765|  Transcript|ENST00000405598|      protein_coding|[12, 16]|  null| null| null|         1451|        1259|             420|     [C, Y]|[tGc, tAc]|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|[DONOR_ESS:34, IN...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[splice_region_va...|     LOW| CHEK2|ENSG00000183765|  Transcript|ENST00000416671|nonsense_mediated...|[12, 16]|  null| null| null|         2033|        null|            null|       null|      null|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[splice_region_va...|     LOW| CHEK2|ENSG00000183765|  Transcript|ENST00000417588|nonsense_mediated...| [9, 13]|  null| null| null|         1168|        null|            null|       null|      null|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[downstream_gene_...|MODIFIER| CHEK2|ENSG00000183765|  Transcript|ENST00000425190|      protein_coding|    null|  null| null| null|         null|        null|            null|       null|      null|              null|    1258|    -1|  [cds_end_NF]|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[downstream_gene_...|MODIFIER| CHEK2|ENSG00000183765|  Transcript|ENST00000433028|nonsense_mediated...|    null|  null| null| null|         null|        null|            null|       null|      null|              null|    1192|    -1|          null|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[splice_region_va...|     LOW| CHEK2|ENSG00000183765|  Transcript|ENST00000433728|nonsense_mediated...| [9, 13]|  null| null| null|         1197|        null|            null|       null|      null|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[stop_gained, spl...|    HIGH| CHEK2|ENSG00000183765|  Transcript|ENST00000434810|      protein_coding| [6, 10]|  null| null| null|          490|         491|             164|     [W, *]|[tGg, tAg]|              null|    null|    -1|[cds_start_NF]|         HGNC|  16627|  HC|      null|     null|[GERP_DIST:673.49...|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[downstream_gene_...|MODIFIER| CHEK2|ENSG00000183765|  Transcript|ENST00000439346|nonsense_mediated...|    null|  null| null| null|         null|        null|            null|       null|      null|              null|    4054|    -1|[cds_start_NF]|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[downstream_gene_...|MODIFIER| CHEK2|ENSG00000183765|  Transcript|ENST00000447421|      protein_coding|    null|  null| null| null|         null|        null|            null|       null|      null|              null|    1188|    -1|  [cds_end_NF]|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|[splice_region_va...|     LOW| CHEK2|ENSG00000183765|  Transcript|ENST00000448511|nonsense_mediated...|[10, 14]|  null| null| null|         1149|        null|            null|       null|      null|              null|    null|    -1|          null|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
|        22|29091697|29091698|              C|             [T]|     T|    [intron_variant]|MODIFIER| CHEK2|ENSG00000183765|  Transcript|ENST00000456369|      protein_coding|    null|[4, 5]| null| null|         null|        null|            null|       null|      null|              null|    null|    -1|[cds_start_NF]|         HGNC|  16627|null|      null|     null|                null|[[HG00096, true, ...|
```

Fig. 6.16: The annotated DataFrame `annotated_variants_df` with expanded subfields of the exploded `INFO_CSQ`

6.3.5 Other Improvements

Glow 0.3.0 also includes optimized implementations of the linear and logistic regression functions, resulting in ~50% performance improvements. See the documentation at *Linear regression* and *Logistic regression*.

Furthermore, the new release supports Scala 2.12 in addition to Scala 2.11. The maven artifacts for both Scala versions are available on Maven Central.

6.3.6 Try It!

Try Glow 0.3.0 and its new features here.

6.4 Streamlining Variant Normalization on Large Genomic Datasets

Author: Kiavash Kianfar
November 20, 2019

Many research and drug development projects in the genomics world involve large genomic variant data sets, the volume of which has been growing exponentially over the past decade. However, the tools to extract, transform, load (ETL) and analyze these data sets have not kept pace with this growth. Single-node command line tools or scripts are very inefficient in handling terabytes of genomics data in these projects. In October of this year, Databricks and the Regeneron Genetics Center partnered to introduce the open-source project Glow, which provides powerful genomics tools based on Apache Spark in order to address this issue.

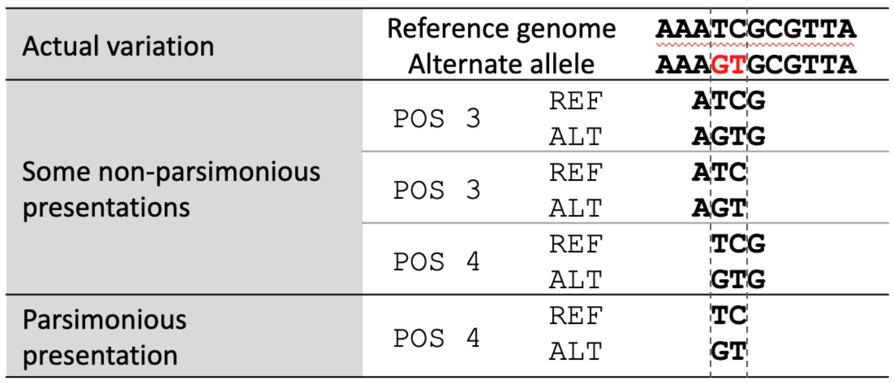
In large cross-team research or drug discovery projects, computational biologists and bioinformaticians usually need to merge very large variant call sets in order to perform downstream analyses. In a prior post, we showcased the power and simplicity of Glow in ETL and merging of variant call sets from different sources using Glow's VCF and BGEN Data Sources at unprecedented scales. Differently sourced variant call sets impose another major challenge. It is not uncommon for these sets to be generated by different variant calling tools and methods. Consequently, the same genomic variant may be represented differently (in terms of genomic position and alleles) across different call sets. These discrepancies in variant representation must be resolved before any further analysis on the data. This is critical for the following reasons:

1. To avoid incorrect bias in the results of downstream analysis on the merged set of variants or waste of analysis effort on seemingly new variants due to lack of normalization, which are in fact redundant (see Tan et al. for examples of this redundancy in 1000 Genome Project variant calls and dbSNP)
2. To ensure that the merged data set and its post-analysis derivations are compatible and comparable with other public and private variant databases.

This is achieved by what is referred to as variant normalization, a process that ensures the same variant is represented identically across different data sets. Performing variant normalization on terabytes of variant data in large projects using popular single-node tools can become quite a challenge as the acceptable input and output of these tools are the flat file formats that are commonly used to store variant calls (such as VCF and BGEN). To address this issue, we introduced the variant normalization transformation into Glow, which directly acts on a Spark Dataframe of variants to generate a DataFrame of normalized variants, harnessing the power of Spark to normalize variants from hundreds of thousands of samples in a fast and scalable manner with just a single line of Python or Scala code. Before addressing our normalizer, let us have a slightly more technical look at what variant normalization actually does.

6.4.1 What does variant normalization do?

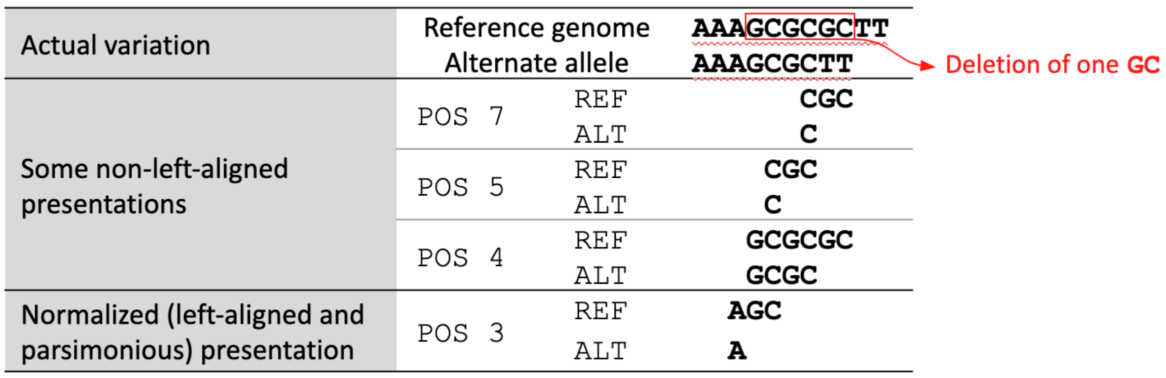
Variant normalization ensures that the representation of a variant is both “parsimonious” and “left-aligned.” A variant is parsimonious if it is represented in as few nucleotides as possible without reducing the length of any allele to zero. An example is given in Fig. 6.17.



Actual variation	Reference genome		AAATCGCGTTA
	Alternate allele		AAAGTGCGTTA
Some non-parsimonious presentations	POS 3	REF	ATCG
		ALT	AGTG
	POS 3	REF	ATC
		ALT	AGT
	POS 4	REF	TCG
		ALT	GTG
Parsimonious presentation	POS 4	REF	TC
		ALT	GT

Fig. 6.17: Variant parsimony

A variant is left-aligned if its position cannot be shifted to the left while keeping the length of all its alleles the same. An example is given in Fig. 6.18.



Actual variation	Reference genome		AAAGCGCGCTT
	Alternate allele		AAAGCGCTT
Some non-left-aligned presentations	POS 7	REF	CGC
		ALT	C
	POS 5	REF	CGC
		ALT	C
	POS 4	REF	GCGCGC
		ALT	GCGC
Normalized (left-aligned and parsimonious) presentation	POS 3	REF	AGC
		ALT	A

Deletion of one GC

Fig. 6.18: Left-aligned variant

Tan et al. have proved that normalization results in uniqueness. In other words, two variants have different normalized representations if and only if they are actually different variants.

6.4.2 Variant normalization in Glow

We have introduced the `normalize_variants` transformer into Glow (Fig. 6.19). After ingesting variant calls into a Spark DataFrame using the VCF, BGEN or Delta readers, a user can call a single line of Python or Scala code to normalize all variants. This generates another DataFrame in which all variants are presented in their normalized form. The normalized DataFrame can then be used for downstream analyses like a GWAS using our built-in regression functions or an efficiently-parallelized GWAS tool.

The `normalize_variants` transformer brings unprecedented scalability and simplicity to this important upstream process, hence is yet another reason why Glow and Databricks UAP for Genomics are ideal platforms for biobank-

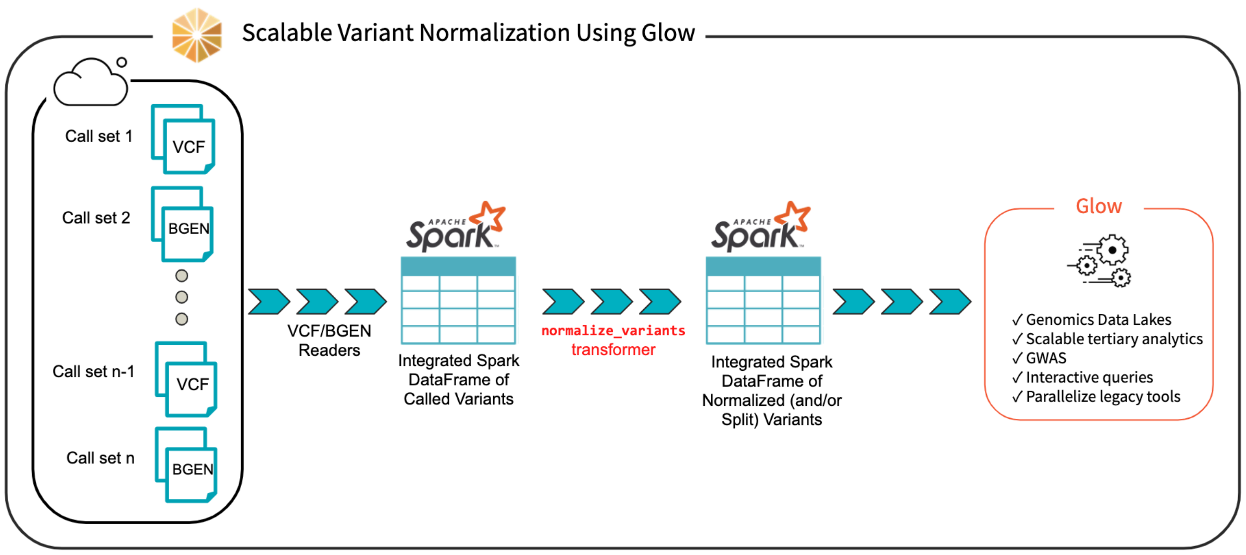

Scalable Variant Normalization Using Glow
Call set 1
VCF
Call set 2
BGEN
Call set n-1
VCF
Call set n
BGEN
VCF/BGEN Readers
Integrated Spark DataFrame of Called Variants
normalize_variants transformer
Integrated Spark DataFrame of Normalized (and/or Split) Variants
Glow
✓ Genomics Data Lakes
✓ Scalable tertiary analytics
✓ GWAS
✓ Interactive queries
✓ Parallelize legacy tools

Fig. 6.19: Scalable Variant Normalization Using Glow

scale genomic analyses, e.g., association studies between genetic variations and diseases across cohorts of hundreds of thousands of individuals.

6.4.3 The underlying normalization algorithm and its accuracy

There are several single-node tools for variant normalization that use different normalization algorithms. Widely used tools for variant normalization include vt normalize, bcftools norm, and the GATK's LeftAlignAndTrimVariants.

Based on our own investigation and also as indicated by *Bayat et al.* and *Tan et al.*, the GATK's LeftAlignAndTrimVariants algorithm frequently fails to completely left-align some variants. For example, we noticed that on the test_left_align_hg38.vcf test file from GATK itself, applying LeftAlignAndTrimVariants results in an incorrect normalization of 3 of the 16 variants in the file, including the variants at positions `chr20:63669973`, `chr20:64012187`, and `chr21:13255301`. These variants are normalized correctly using `vt normalize` and `bcftools norm`.

Consequently, in our `normalize_variants` transformer, we used an improved version of the `bcftools norm` or `vt normalize` algorithms, which are similar in fundamentals. For a given variant, we start by right-trimming all the alleles of the variant as long as their rightmost nucleotides are the same. If the length of any allele reaches zero, we left-append it with a fixed block of nucleotides from the reference genome (the nucleotides are added in blocks as opposed to one-by-one to limit the number of referrals to the reference genome). When right-trimming is terminated, a potential left-trimming is performed to eliminate the leftmost nucleotides common to all alleles (possibly generated by prior left-appendings). The start, end, and alleles of the variants are updated appropriately during this process.

We benchmarked the accuracy of our normalization algorithm against `vt normalize` and `bcftools norm` on multiple test files and validated that our results match the results of these tools.

6.4.4 Optional splitting

Our `normalize_variants` transformer can optionally split multiallelic variants to biallelics. This is controlled by the mode option that can be supplied to this transformer. The possible values for the mode option are as follows: `normalize` (default), which performs normalization only, `split_and_normalize`, which splits multiallelic variants to biallelic ones before performing normalization, and `split`, which only splits multiallelics without doing any normalization.

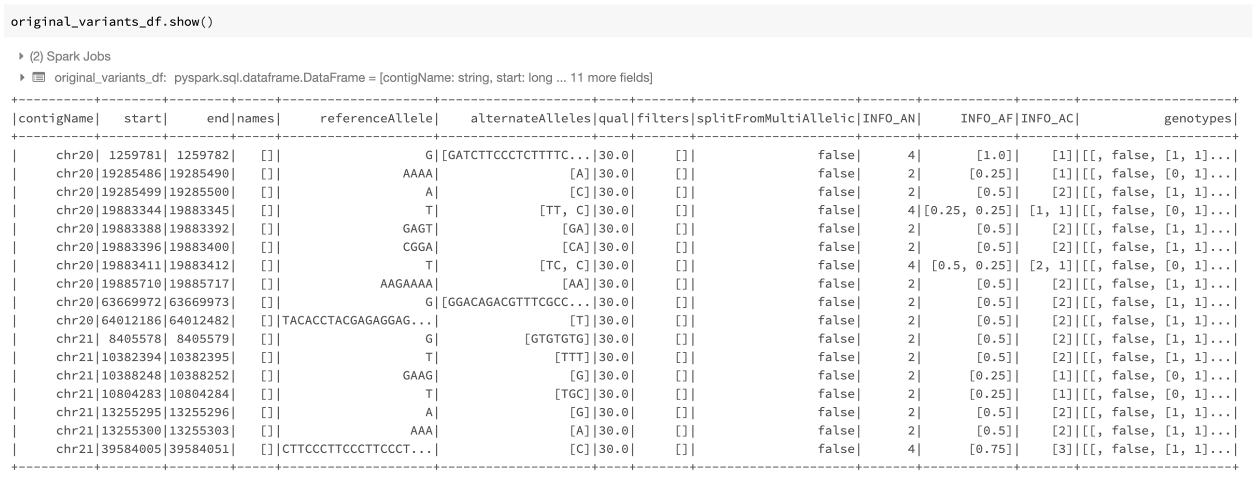
The splitting logic of our transformer is the same as the splitting logic followed by GATK's LeftAlignAndTrimVariants tool using `--splitMultiallelics` option. More precisely, in case of splitting multiallelic variants loaded from VCF files, this transformer recalculates the `GT` blocks for the resulting biallelic variants if possible, and drops all `INFO` fields, except for `AC`, `AN`, and `AF`. These three fields are imputed based on the newly calculated `GT` blocks, if any exists, otherwise, these fields are dropped as well.

6.4.5 Using the transformer

Here, we briefly demonstrate how using Glow very large variant call sets can be normalized and/or split. First, VCF and/or BGEN files can be read into a Spark DataFrame as demonstrated in a prior post. This is shown in Python for the set of VCF files contained in a folder named `/databricks-datasets/genomics/call-sets`:

```
original_variants_df = spark.read\
  .format("vcf")\
  .option("includeSampleIds", False)\
  .load("/databricks-datasets/genomics/call-sets")
```

An example of the DataFrame original_variants_df is shown in Fig. 6.20.



```
original_variants_df.show()
```

(2) Spark Jobs
original_variants_df: pyspark.sql.dataframe.DataFrame = [contigName: string, start: long ... 11 more fields]

```
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
|contigName|   start|     end|names|     referenceAllele|    alternateAlleles|qual|filters|splitFromMultiAllelic|INFO_AN|     INFO_AF|INFO_AC|           genotypes|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
|     chr20| 1259781| 1259782|   []|                   G|[GATCTTCCCTCTTTTC...|30.0|     []|                false|      4|       [1.0]|    [1]|[[, false, [1, 1]...|
|     chr20|19285486|19285490|   []|                AAAA|                 [A]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr20|19285499|19285500|   []|                   A|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883344|19883345|   []|                   T|             [TT, C]|30.0|     []|                false|      4|[0.25, 0.25]| [1, 1]|[[, false, [0, 1]...|
|     chr20|19883388|19883392|   []|                GAGT|                [GA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883396|19883400|   []|                CGGA|                [CA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883411|19883412|   []|                   T|             [TC, C]|30.0|     []|                false|      4| [0.5, 0.25]| [2, 1]|[[, false, [0, 1]...|
|     chr20|19885710|19885717|   []|             AAGAAAA|                [AA]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|63669972|63669973|   []|                   G|[GGACAGACGTTTCGCC...|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|64012186|64012482|   []|TACACCTACGAGAGGAG...|                 [T]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21| 8405578| 8405579|   []|                   G|           [GTGTGTG]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|10382394|10382395|   []|                   T|               [TTT]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|10388248|10388252|   []|                GAAG|                 [G]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr21|10804283|10804284|   []|                   T|               [TGC]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr21|13255295|13255296|   []|                   A|                 [G]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|13255300|13255303|   []|                 AAA|                 [A]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|39584005|39584051|   []|CTTCCCTTCCCTTCCCT...|                 [C]|30.0|     []|                false|      4|      [0.75]|    [3]|[[, false, [1, 1]...|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
```

Fig. 6.20: The variant DataFrame original_variants_df

The variants can then be normalized using the `normalize_variants` transformer as follows:

```
import glow

ref_genome_path = '/mnt/dbnucleus/dbgenomics/grch38/data/GRCh38.fa'

normalized_variants_df = glow.transform(\
  "normalize_variants",\
  original_variants_df,\
```

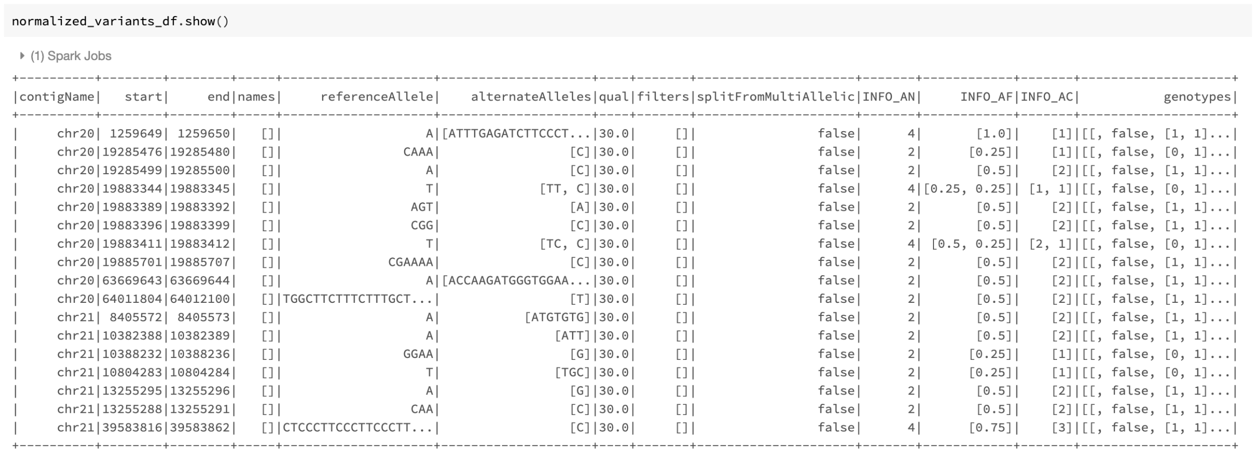
(continues on next page)

(continued from previous page)

```
  reference_genome_path=ref_genome_path\
)
```

Note that normalization requires the reference genome `.fasta` or `.fa` file, which is provided using the `reference_genome_path` option. The `.dict` and `.fai` files must accompany the reference genome file in the same folder (read more about these file formats here).

Our example Dataframe after normalization can be seen in Fig. 6.21.



```
normalized_variants_df.show()
```

▸ (1) Spark Jobs

```
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
|contigName|   start|     end|names|     referenceAllele|    alternateAlleles|qual|filters|splitFromMultiAllelic|INFO_AN|     INFO_AF|INFO_AC|           genotypes|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
|     chr20| 1259649| 1259650|   []|                   A|[ATTTGAGATCTTCCCT...|30.0|     []|                false|      4|       [1.0]|    [1]|[[, false, [1, 1]...|
|     chr20|19285476|19285480|   []|                CAAA|                 [C]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr20|19285499|19285500|   []|                   A|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883344|19883345|   []|                   T|             [TT, C]|30.0|     []|                false|      4|[0.25, 0.25]| [1, 1]|[[, false, [0, 1]...|
|     chr20|19883389|19883392|   []|                 AGT|                 [A]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883396|19883399|   []|                 CGG|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883411|19883412|   []|                   T|             [TC, C]|30.0|     []|                false|      4| [0.5, 0.25]| [2, 1]|[[, false, [0, 1]...|
|     chr20|19885701|19885707|   []|              CGAAAA|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|63669643|63669644|   []|                   A|[ACCAAGATGGGTGGAA...|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|64011804|64012100|   []|TGGCTTCTTTCTTTGCT...|                 [T]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21| 8405572| 8405573|   []|                   A|           [ATGTGTG]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|10382388|10382389|   []|                   A|               [ATT]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|10388232|10388236|   []|                GGAA|                 [G]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr21|10804283|10804284|   []|                   T|               [TGC]|30.0|     []|                false|      2|      [0.25]|    [1]|[[, false, [0, 1]...|
|     chr21|13255295|13255296|   []|                   A|                 [G]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|13255288|13255291|   []|                 CAA|                 [C]|30.0|     []|                false|      2|       [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|39583816|39583862|   []|CTCCCTTCCCTTCCCTT...|                 [C]|30.0|     []|                false|      4|      [0.75]|    [3]|[[, false, [1, 1]...|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+------------+-------+--------------------+
```

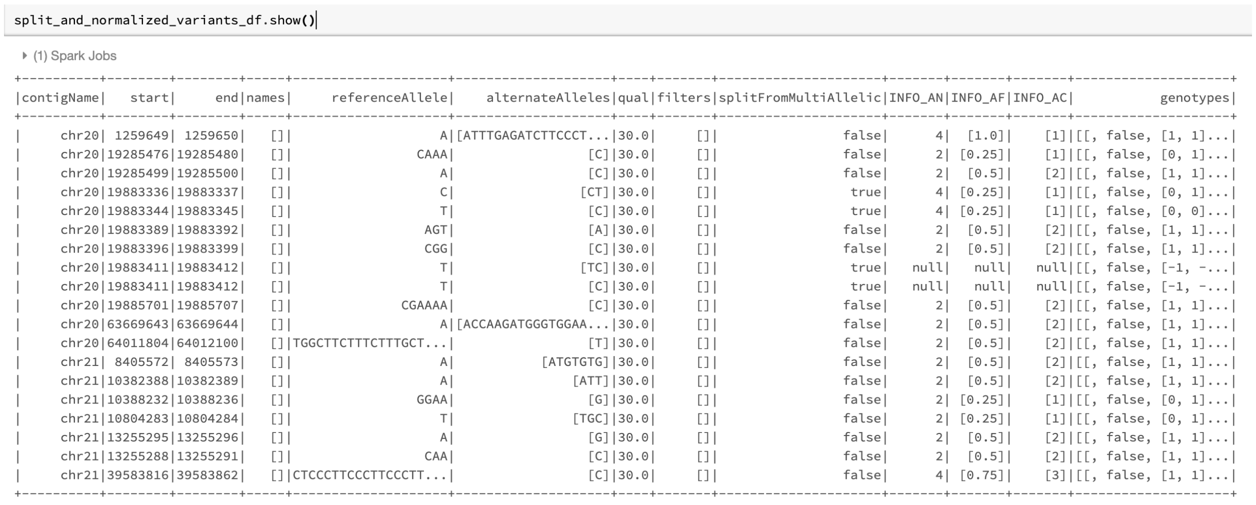
Fig. 6.21: The `normalized_variants_df` DataFrame obtained after applying `normalize_variants` transformer on `original_variants_df`. Notice that several variants are normalized and their start, end, and alleles have changed accordingly.

By default, the transformer normalizes each variant without splitting the multiallelic variants before normalization as seen in Fig. 6.21. By setting the mode option to `split_and_normalize`, nothing changes for biallelic variants, but the multiallelic variants are first split to the appropriate number of biallelics and the resulting biallelics are normalized. This can be done as follows:

```
split_and_normalized_variants_df = glow.transform(\
  "normalize_variants",\
  original_variants_df,\
  reference_genome_path=ref_genome_path,\
  mode=“split_and_normalize”
)
```

The resulting DataFrame looks like Fig. 6.22.

As mentioned before, the transformer can also be used only for splitting of multiallelics without doing any normalization by setting the mode option to `split`.



```
split_and_normalized_variants_df.show()
```

```
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+-------+-------+--------------------+
|contigName|   start|     end|names|     referenceAllele|    alternateAlleles|qual|filters|splitFromMultiAllelic|INFO_AN|INFO_AF|INFO_AC|           genotypes|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+-------+-------+--------------------+
|     chr20| 1259649| 1259650|   []|                   A|[ATTTGAGATCTTCCCT...|30.0|     []|                false|      4|  [1.0]|    [1]|[[, false, [1, 1]...|
|     chr20|19285476|19285480|   []|                CAAA|                 [C]|30.0|     []|                false|      2| [0.25]|    [1]|[[, false, [0, 1]...|
|     chr20|19285499|19285500|   []|                   A|                 [C]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883336|19883337|   []|                   C|                [CT]|30.0|     []|                 true|      4| [0.25]|    [1]|[[, false, [0, 1]...|
|     chr20|19883344|19883345|   []|                   T|                 [C]|30.0|     []|                 true|      4| [0.25]|    [1]|[[, false, [0, 0]...|
|     chr20|19883389|19883392|   []|                 AGT|                 [A]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883396|19883399|   []|                 CGG|                 [C]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|19883411|19883412|   []|                   T|                [TC]|30.0|     []|                 true|   null|   null|   null|[[, false, [-1, -...|
|     chr20|19883411|19883412|   []|                   T|                 [C]|30.0|     []|                 true|   null|   null|   null|[[, false, [-1, -...|
|     chr20|19885701|19885707|   []|              CGAAAA|                 [C]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|63669643|63669644|   []|                   A|[ACCAAGATGGGTGGAA...|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr20|64011804|64012100|   []|TGGCTTCTTTCTTTGCT...|                 [T]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21| 8405572| 8405573|   []|                   A|           [ATGTGTG]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|10382388|10382389|   []|                   A|               [ATT]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|10388232|10388236|   []|                GGAA|                 [G]|30.0|     []|                false|      2| [0.25]|    [1]|[[, false, [0, 1]...|
|     chr21|10804283|10804284|   []|                   T|               [TGC]|30.0|     []|                false|      2| [0.25]|    [1]|[[, false, [0, 1]...|
|     chr21|13255295|13255296|   []|                   A|                 [G]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|13255288|13255291|   []|                 CAA|                 [C]|30.0|     []|                false|      2|  [0.5]|    [2]|[[, false, [1, 1]...|
|     chr21|39583816|39583862|   []|CTCCCTTCCCTTCCCTT...|                 [C]|30.0|     []|                false|      4| [0.75]|    [3]|[[, false, [1, 1]...|
+----------+--------+--------+-----+--------------------+--------------------+----+-------+---------------------+-------+-------+-------+--------------------+
```

Fig. 6.22: The `split_and_normalized_variants_df` DataFrame after applying normalize_variants transformer with `mode=split_and_normalize` on `original_variants_df`. Notice that for example the triallelic variant `chr20,start=19883344,end=19883345,REF=T,ALT=[TT,C]` of `original_variants_df` has been split into two biallelic variants and then normalized resulting in two normalized biallelic variants `chr20,start=19883336, end=19883337,REF=C,ALT=CT` and `chr20,start=19883344,end=19883345,REF=T,ALT=C`.

6.4.6 Summary

Using Glow `normalize_variants` transformer, computational biologists and bioinformaticians can normalize very large variant datasets of hundreds of thousands of samples in a fast and scalable manner. Differently sourced call sets can be ingested and merged using VCF and/or BGEN readers, normalization can be performed using this transformer in a just a single line of code. The transformer can optionally perform splitting of multiallelic variants to biallelics as well.

6.4.7 Try it!

Our normalize_variants transformer makes it easy to normalize (and split) large variant datasets with a very small amount of code . Learn more about other feature of Glow here.

6.4.8 References

Arash Bayat, Bruno Gaëta, Aleksandar Ignjatovic, Sri Parameswaran, Improved VCF normalization for accurate VCF comparison, Bioinformatics, Volume 33, Issue 7, 2017, Pages 964–970

Adrian Tan, Gonçalo R. Abecasis, Hyun Min Kang, Unified representation of genetic variants, Bioinformatics, Volume 31, Issue 13, 2015, Pages 2202–2204

CHAPTER

SEVEN

ADDITIONAL RESOURCES

7.1 Databricks notebooks

Most of the code in the Databricks notebooks can be run on Spark and Glow alone, but some functions are only available on Databricks.

7.1.1 New to Databricks? Try Glow on Databricks for Free!

The Databricks Community Edition is free of charge. Follow our instructions to set up a Databricks Community Edition workspace and try the Glow documentation notebooks.

7.2 External blog posts

- Scaling Genomic Workflows with Spark SQL BGEN and VCF Readers
- Parallelizing SAIGE Across Hundreds of Cores
  - Parallelize SAIGE using Glow and the Pipe Transformer
- Accurately Building Genomic Cohorts at Scale with Delta Lake and Spark SQL
  - Joint genotyping with Glow and Databricks
- Introducing Glow: an open-source toolkit for large-scale genomic analysis

CHAPTER

EIGHT

PYTHON API

Glow's Python API is designed to work seamlessly with PySpark and other tools in the Spark ecosystem. The functions here work with normal PySpark DataFrames and columns. You can access functions in any module from the top-level `glow` import.

8.1 Glow Top-Level Functions

`glow.`**`register`**`(`*session*, *new_session=True*`)`

Register SQL extensions and py4j converters for a Spark session.

Parameters

- **`session`** (`SparkSession`) – Spark session
- **`new_session`** (`bool`) – If `True`, create a new Spark session using `session.newSession()` before registering extensions. This may be necessary if you're using functions that register new analysis rules. The new session has isolated UDFs, configurations, and temporary tables, but shares the existing `SparkContext` and cached data.

Example

```
>>> import glow
>>> spark = glow.register(spark)
```

Return type `SparkSession`

`glow.`**`transform`**`(`*operation*, *df*, *arg_map=None*, ***kwargs*`)`

Apply a named transformation to a DataFrame of genomic data. All parameters apart from the input data and its schema are provided through the case-insensitive options map.

There are no bounds on what a transformer may do. For instance, it's legal for a transformer to materialize the input DataFrame.

Parameters

- **`operation`** (`str`) – Name of the operation to perform
- **`df`** (`DataFrame`) – The input DataFrame
- **`arg_map`** (`Optional[Dict[str, Any]]`) – A string -> any map of arguments
- **`kwargs`** (`Any`) – Named arguments. If the arg_map is not specified, transformer args will be pulled from these keyword args.

Example

```
>>> df = spark.read.format('vcf').load('test-data/1kg_sample.vcf')
>>> piped_df = glow.transform('pipe', df, cmd=["cat"], input_formatter='vcf',␣
↪output_formatter='vcf', in_vcf_header='infer')
```

Return type `DataFrame`

Returns The transformed DataFrame

8.2 PySpark Functions

Glow includes a number of functions that operate on PySpark columns. These functions are interoperable with functions provided by PySpark or other libraries.

`glow.`**`add_struct_fields`**(*struct*, **fields*)

Adds fields to a struct.

Added in version 0.3.0.

Examples

```
>>> df = spark.createDataFrame([Row(struct=Row(a=1))])
>>> df.select(glow.add_struct_fields('struct', lit('b'), lit(2)).alias('struct')).
↪collect()
[Row(struct=Row(a=1, b=2))]
```

Parameters

- **`struct`** (`Union[Column, str]`) – The struct to which fields will be added
- **`fields`** (`Union[Column, str]`) – The new fields to add. The arguments must alternate between string-typed literal field names and field values.

Return type `Column`

Returns A struct consisting of the input struct and the added fields

`glow.`**`array_summary_stats`**(*arr*)

Computes the minimum, maximum, mean, standard deviation for an array of numerics.

Added in version 0.3.0.

Examples

```
>>> df = spark.createDataFrame([Row(arr=[1, 2, 3])])
>>> df.select(glow.expand_struct(glow.array_summary_stats('arr'))).collect()
[Row(mean=2.0, stdDev=1.0, min=1.0, max=3.0)]
```

Parameters **`arr`** (`Union[Column, str]`) – An array of any numeric type

Return type `Column`

Returns A struct containing double `mean`, `stdDev`, `min`, and `max` fields

`glow.`**`array_to_dense_vector`**`(`*arr*`)`

Converts an array of numerics into a `spark.ml` `DenseVector`.

Added in version 0.3.0.

Examples

```
>>> from pyspark.ml.linalg import DenseVector
>>> df = spark.createDataFrame([Row(arr=[1, 2, 3])])
>>> df.select(glow.array_to_dense_vector('arr').alias('v')).collect()
[Row(v=DenseVector([1.0, 2.0, 3.0]))]
```

Parameters **arr** (`Union`[`Column`, `str`]) – The array of numerics

Return type `Column`

Returns A `spark.ml` `DenseVector`

`glow.`**`array_to_sparse_vector`**`(`*arr*`)`

Converts an array of numerics into a `spark.ml` `SparseVector`.

Added in version 0.3.0.

Examples

```
>>> from pyspark.ml.linalg import SparseVector
>>> df = spark.createDataFrame([Row(arr=[1, 0, 2, 0, 3, 0])])
>>> df.select(glow.array_to_sparse_vector('arr').alias('v')).collect()
[Row(v=SparseVector(6, {0: 1.0, 2: 2.0, 4: 3.0}))]
```

Parameters **arr** (`Union`[`Column`, `str`]) – The array of numerics

Return type `Column`

Returns A `spark.ml` `SparseVector`

`glow.`**`call_summary_stats`**`(`*genotypes*`)`

Computes call summary statistics for an array of genotype structs. See *Variant Quality Control* for more details.

Added in version 0.3.0.

Examples

```
>>> schema = 'genotypes: array<struct<calls: array<int>>>'
>>> df = spark.createDataFrame([Row(genotypes=[Row(calls=[0, 0]), Row(calls=[1, 0]),
↪ Row(calls=[1, 1])])], schema)
>>> df.select(glow.expand_struct(glow.call_summary_stats('genotypes'))).collect()
[Row(callRate=1.0, nCalled=3, nUncalled=0, nHet=1, nHomozygous=[1, 1], nNonRef=2,␣
↪nAllelesCalled=6, alleleCounts=[3, 3], alleleFrequencies=[0.5, 0.5])]
```

Parameters **genotypes** (`Union`[`Column`, `str`]) – The array of genotype structs with `calls` field

Return type `Column`

Returns A struct containing `callRate`, `nCalled`, `nUncalled`, `nHet`, `nHomozygous`, `nNonRef`, `nAllelesCalled`, `alleleCounts`, `alleleFrequencies` fields. See *Variant Quality Control*.

`glow.`**`dp_summary_stats`**`(`*genotypes*`)`

Computes summary statistics for the depth field from an array of genotype structs. See *Variant Quality Control*.

Added in version 0.3.0.

Examples

```
>>> df = spark.createDataFrame([Row(genotypes=[Row(depth=1), Row(depth=2),␣
↪Row(depth=3)])], 'genotypes: array<struct<depth: int>>')
>>> df.select(glow.expand_struct(glow.dp_summary_stats('genotypes'))).collect()
[Row(mean=2.0, stdDev=1.0, min=1.0, max=3.0)]
```

Parameters **`genotypes`** (`Union`[`Column`, `str`]) – An array of genotype structs with `depth` field

Return type `Column`

Returns A struct containing `mean`, `stdDev`, `min`, and `max` of genotype depths

`glow.`**`expand_struct`**`(`*struct*`)`

Promotes fields of a nested struct to top-level columns similar to using `struct.*` from SQL, but can be used in more contexts.

Added in version 0.3.0.

Examples

```
>>> df = spark.createDataFrame([Row(struct=Row(a=1, b=2))])
>>> df.select(glow.expand_struct(col('struct'))).collect()
[Row(a=1, b=2)]
```

Parameters **`struct`** (`Union`[`Column`, `str`]) – The struct to expand

Return type `Column`

Returns Columns corresponding to fields of the input struct

`glow.`**`explode_matrix`**`(`*matrix*`)`

Explodes a `spark.ml` `Matrix` (sparse or dense) into multiple arrays, one per row of the matrix.

Added in version 0.3.0.

Examples

```
>>> from pyspark.ml.linalg import DenseMatrix
>>> m = DenseMatrix(numRows=3, numCols=2, values=[1, 2, 3, 4, 5, 6])
>>> df = spark.createDataFrame([Row(matrix=m)])
>>> df.select(glow.explode_matrix('matrix').alias('row')).collect()
[Row(row=[1.0, 4.0]), Row(row=[2.0, 5.0]), Row(row=[3.0, 6.0])]
```

Parameters **matrix** (Union[Column, str]) – The sparl.ml Matrix to explode

Return type Column

Returns An array column in which each row is a row of the input matrix

glow.**genotype_states**(*genotypes*)

Gets the number of alternate alleles for an array of genotype structs. Returns -1 if there are any -1 s (no-calls) in the calls array.

Added in version 0.3.0.

Examples

```
>>> genotypes = [
... Row(calls=[1, 1]),
... Row(calls=[1, 0]),
... Row(calls=[0, 0]),
... Row(calls=[-1, -1])]
>>> df = spark.createDataFrame([Row(genotypes=genotypes)], 'genotypes: array<struct
↪<calls: array<int>>>')
>>> df.select(glow.genotype_states('genotypes').alias('states')).collect()
[Row(states=[2, 1, 0, -1])]
```

Parameters **genotypes** (Union[Column, str]) – An array of genotype structs with calls field

Return type Column

Returns An array of integers containing the number of alternate alleles in each call array

glow.**gq_summary_stats**(*genotypes*)

Computes summary statistics about the genotype quality field for an array of genotype structs. See *Variant Quality Control*.

Added in version 0.3.0.

Examples

```
>>> genotypes = [
... Row(conditionalQuality=1),
... Row(conditionalQuality=2),
... Row(conditionalQuality=3)]
>>> df = spark.createDataFrame([Row(genotypes=genotypes)], 'genotypes: array<struct
↪<conditionalQuality: int>>')
```

(continues on next page)

(continued from previous page)

```
>>> df.select(glow.expand_struct(glow.gq_summary_stats('genotypes'))).collect()
[Row(mean=2.0, stdDev=1.0, min=1.0, max=3.0)]
```

Parameters **genotypes** (Union[Column, str]) – The array of genotype structs with `conditionalQuality` field

Return type `Column`

Returns A struct containing `mean`, `stdDev`, `min`, and `max` of genotype qualities

`glow.`**`hard_calls`**(*probabilities*, *numAlts*, *phased*, *threshold=None*)

Converts an array of probabilities to hard calls. The probabilities are assumed to be diploid. See *Variant data transformations* for more details.

Added in version 0.3.0.

Examples

```
>>> df = spark.createDataFrame([Row(probs=[0.95, 0.05, 0.0])])
>>> df.select(glow.hard_calls('probs', numAlts=lit(1), phased=lit(False)).alias(
↪'calls')).collect()
[Row(calls=[0, 0])]
>>> df = spark.createDataFrame([Row(probs=[0.05, 0.95, 0.0])])
>>> df.select(glow.hard_calls('probs', numAlts=lit(1), phased=lit(False)).alias(
↪'calls')).collect()
[Row(calls=[0, 1])]
>>> # Use the threshold parameter to change the minimum probability required for a␣
↪call
>>> df = spark.createDataFrame([Row(probs=[0.05, 0.95, 0.0])])
>>> df.select(glow.hard_calls('probs', numAlts=lit(1), phased=lit(False),␣
↪threshold=0.99).alias('calls')).collect()
[Row(calls=[-1, -1])]
```

Parameters

- **probabilities** (Union[Column, str]) – The array of probabilities to convert
- **numAlts** (Union[Column, str]) – The number of alternate alleles
- **phased** (Union[Column, str]) – Whether the probabilities are phased. If phased, we expect one `2 * numAlts` values in the probabilities array. If unphased, we expect one probability per possible genotype.
- **threshold** (Optional[float]) – The minimum probability to make a call. If no probability falls into the range of `[0, 1 - threshold]` or `[threshold, 1]`, a no-call (represented by `-1` s) will be emitted. If not provided, this parameter defaults to `0.9`.

Return type `Column`

Returns An array of hard calls

`glow.`**`hardy_weinberg`**(*genotypes*)

Computes statistics relating to the Hardy Weinberg equilibrium. See *Variant Quality Control* for more details.

Added in version 0.3.0.

Examples

```
>>> genotypes = [
... Row(calls=[1, 1]),
... Row(calls=[1, 0]),
... Row(calls=[0, 0])]
>>> df = spark.createDataFrame([Row(genotypes=genotypes)], 'genotypes: array<struct
↪<calls: array<int>>>')
>>> df.select(glow.expand_struct(glow.hardy_weinberg('genotypes'))).collect()
[Row(hetFreqHwe=0.6, pValueHwe=0.7)]
```

Parameters **genotypes** (`Union`[`Column`, `str`]) – The array of genotype structs with `calls` field

Return type `Column`

Returns A struct containing two fields, `hetFreqHwe` (the expected heterozygous frequency according to Hardy-Weinberg equilibrium) and `pValueHwe` (the associated p-value)

`glow.`**`lift_over_coordinates`**(*contigName*, *start*, *end*, *chainFile*, *minMatchRatio=None*)

Performs liftover for the coordinates of a variant. To perform liftover of alleles and add additional metadata, see *Liftover*.

Added in version 0.3.0.

Examples

```
>>> df = spark.read.format('vcf').load('test-data/liftover/unlifted.test.vcf').
↪where('start = 18210071')
>>> chain_file = 'test-data/liftover/hg38ToHg19.over.chain.gz'
>>> reference_file = 'test-data/liftover/hg19.chr20.fa.gz'
>>> df.select('contigName', 'start', 'end').head()
Row(contigName='chr20', start=18210071, end=18210072)
>>> lifted_df = df.select(glow.expand_struct(glow.lift_over_coordinates('contigName
↪', 'start', 'end', chain_file)))
>>> lifted_df.head()
Row(contigName='chr20', start=18190715, end=18190716)
```

Parameters

- **`contigName`** (`Union`[`Column`, `str`]) – The current contig name
- **`start`** (`Union`[`Column`, `str`]) – The current start
- **`end`** (`Union`[`Column`, `str`]) – The current end
- **`chainFile`** (`str`) – Location of the chain file on each node in the cluster
- **`minMatchRatio`** (`Optional`[`float`]) – Minimum fraction of bases that must remap to do liftover successfully. If not provided, defaults to `0.95`.

Return type `Column`

Returns A struct containing `contigName`, `start`, and `end` fields after liftover

`glow.`**`linear_regression_gwas`**(*genotypes*, *phenotypes*, *covariates*)

Performs a linear regression association test optimized for performance in a GWAS setting. See *Linear regression* for details.

Added in version 0.3.0.

Examples

```
>>> from pyspark.ml.linalg import DenseMatrix
>>> phenotypes = [2, 3, 4]
>>> genotypes = [0, 1, 2]
>>> covariates = DenseMatrix(numRows=3, numCols=1, values=[1, 1, 1])
>>> df = spark.createDataFrame([Row(genotypes=genotypes, phenotypes=phenotypes,␣
↪covariates=covariates)])
>>> df.select(glow.expand_struct(glow.linear_regression_gwas('genotypes',
↪'phenotypes', 'covariates'))).collect()
[Row(beta=0.9999999999999998, standardError=1.4901161193847656e-08, pValue=9.
↪486373847239922e-09)]
```

Parameters

- **genotypes** (Union[Column, str]) – A numeric array of genotypes
- **phenotypes** (Union[Column, str]) – A numeric array of phenotypes
- **covariates** (Union[Column, str]) – A spark.ml Matrix of covariates

Return type Column

Returns A struct containing beta, standardError, and pValue fields. See *Linear regression*.

glow.**logistic_regression_gwas**(*genotypes*, *phenotypes*, *covariates*, *test*, *offset=None*)

Performs a logistic regression association test optimized for performance in a GWAS setting. See *Logistic regression* for more details.

Added in version 0.3.0.

Examples

```
>>> from pyspark.ml.linalg import DenseMatrix
>>> phenotypes = [1, 0, 0, 1, 1]
>>> genotypes = [0, 0, 1, 2, 2]
>>> covariates = DenseMatrix(numRows=5, numCols=1, values=[1, 1, 1, 1, 1])
>>> offset = [1, 0, 1, 0, 1]
>>> df = spark.createDataFrame([Row(genotypes=genotypes, phenotypes=phenotypes,␣
↪covariates=covariates, offset=offset)])
>>> df.select(glow.expand_struct(glow.logistic_regression_gwas('genotypes',
↪'phenotypes', 'covariates', 'Firth'))).collect()
[Row(beta=0.7418937644793101, oddsRatio=2.09990848346903, waldConfidenceInterval=[0.
↪2509874689201784, 17.569066925598555], pValue=0.3952193664793294)]
>>> df.select(glow.expand_struct(glow.logistic_regression_gwas('genotypes',
↪'phenotypes', 'covariates', 'LRT'))).collect()
[Row(beta=1.1658962684583645, oddsRatio=3.208797538802116,␣
↪waldConfidenceInterval=[0.29709600522888285, 34.65674887513274], pValue=0.
↪2943946848756769)]
>>> df.select(glow.expand_struct(glow.logistic_regression_gwas('genotypes',
↪'phenotypes', 'covariates', 'Firth', 'offset'))).collect()
[Row(beta=0.8024832156793392, oddsRatio=2.231074294047771,␣
↪waldConfidenceInterval=[0.2540891981649045, 19.590334974925725], pValue=0.
↪3754070658316332)]
```

(continues on next page)

(continued from previous page)

```
>>> df.select(glow.expand_struct(glow.logistic_regression_gwas('genotypes',
↪'phenotypes', 'covariates', 'LRT', 'offset'))).collect()
[Row(beta=1.1996041727573317, oddsRatio=3.3188029900720117,␣
↪waldConfidenceInterval=[0.3071189078535928, 35.863807161497334], pValue=0.
↪2857137988674153)]
```

Parameters

- **genotypes** (Union[Column, str]) – An numeric array of genotypes
- **phenotypes** (Union[Column, str]) – A double array of phenotype values
- **covariates** (Union[Column, str]) – A `spark.ml` `Matrix` of covariates
- **test** (str) – Which logistic regression test to use. Can be `LRT` or `Firth`
- **offset** (Union[Column, str, None]) – An optional double array of offset values. The offset vector is added with coefficient 1 to the linear predictor term X*b.

Return type `Column`

Returns A struct containing `beta`, `oddsRatio`, `waldConfidenceInterval`, and `pValue` fields. See *Logistic regression*.

`glow.`**`mean_substitute`**(*array*, *missingValue=None*)

Substitutes the missing values of a numeric array using the mean of the non-missing values. Any values that are NaN, null or equal to the missing value parameter are considered missing. See *Variant data transformations* for more details.

Added in version 0.4.0.

Examples

```
>>> df = spark.createDataFrame([Row(unsubstituted_values=[float('nan'), None, 0.0,␣
↪1.0, 2.0, 3.0, 4.0])])
>>> df.select(glow.mean_substitute('unsubstituted_values', lit(0.0)).alias(
↪'substituted_values')).collect()
[Row(substituted_values=[2.5, 2.5, 2.5, 1.0, 2.0, 3.0, 4.0])]
>>> df = spark.createDataFrame([Row(unsubstituted_values=[0, 1, 2, 3, -1, None])])
>>> df.select(glow.mean_substitute('unsubstituted_values').alias('substituted_values
↪')).collect()
[Row(substituted_values=[0.0, 1.0, 2.0, 3.0, 1.5, 1.5])]
```

Parameters

- **array** (Union[Column, str]) – A numeric array that may contain missing values
- **missingValue** (Union[Column, str, None]) – A value that should be considered missing. If not provided, this parameter defaults to `-1`.

Return type `Column`

Returns A numeric array with substituted missing values

`glow.`**`normalize_variant`**(*contigName*, *start*, *end*, *refAllele*, *altAlleles*, *refGenomePathString*)

Normalizes the variant with a behavior similar to vt normalize or bcftools norm. Creates a StructType column including the normalized `start`, `end`, `referenceAllele` and `alternateAlleles` fields (whether they are changed or unchanged as the result of normalization) as well as a StructType field called `normalizationStatus` that contains the following fields:

> `changed`: A boolean field indicating whether the variant data was changed as a result of normalization
>
> `errorMessage`: An error message in case the attempt at normalizing the row hit an error. In this case, the `changed` field will be set to `false`. If no errors occur, this field will be `null`.

In case of an error, the `start`, `end`, `referenceAllele` and `alternateAlleles` fields in the generated struct will be `null`.

Added in version 0.3.0.

Examples

```
>>> df = spark.read.format('vcf').load('test-data/variantsplitternormalizer-test/
↪test_left_align_hg38_altered.vcf')
>>> ref_genome = 'test-data/variantsplitternormalizer-test/Homo_sapiens_assembly38.
↪20.21_altered.fasta'
>>> df.select('contigName', 'start', 'end', 'referenceAllele', 'alternateAlleles').
↪head()
Row(contigName='chr20', start=400, end=401, referenceAllele='G', alternateAlleles=[
↪'GATCTTCCCTCTTTTCTAATATAAACACATAAAGCTCTGTTTCCTTCTAGGTAACTGGTTTGAG'])
>>> normalized_df = df.select('contigName', glow.expand_struct(glow.normalize_
↪variant('contigName', 'start', 'end', 'referenceAllele', 'alternateAlleles', ref_
↪genome)))
>>> normalized_df.head()
Row(contigName='chr20', start=268, end=269, referenceAllele='A', alternateAlleles=[
↪'ATTTGAGATCTTCCCTCTTTTCTAATATAAACACATAAAGCTCTGTTTCCTTCTAGGTAACTGG'],␣
↪normalizationStatus=Row(changed=True, errorMessage=None))
```

Parameters

- **`contigName`** (`Union`[`Column`, `str`]) – The current contig name
- **`start`** (`Union`[`Column`, `str`]) – The current start
- **`end`** (`Union`[`Column`, `str`]) – The current end
- **`refAllele`** (`Union`[`Column`, `str`]) – The current reference allele
- **`altAlleles`** (`Union`[`Column`, `str`]) – The current array of alternate alleles
- **`refGenomePathString`** (`str`) – A path to the reference genome `.fasta` file. The `.fasta` file must be accompanied with a `.fai` index file in the same folder.

Return type `Column`

Returns A struct as explained above

`glow.`**`sample_call_summary_stats`**(*genotypes*, *refAllele*, *alternateAlleles*)

Computes per-sample call summary statistics. See *Sample Quality Control* for more details.

Added in version 0.3.0.

Examples

```
>>> sites = [
... {'refAllele': 'C', 'alternateAlleles': ['G'], 'genotypes': [{'sampleId':
↪'NA12878', 'calls': [0, 0]}]},
... {'refAllele': 'A', 'alternateAlleles': ['G'], 'genotypes': [{'sampleId':
↪'NA12878', 'calls': [1, 1]}]},
... {'refAllele': 'AG', 'alternateAlleles': ['A'], 'genotypes': [{'sampleId':
↪'NA12878', 'calls': [1, 0]}]}]
>>> df = spark.createDataFrame(sites, 'refAllele: string, alternateAlleles: array
↪<string>, genotypes: array<struct<sampleId: string, calls: array<int>>>')
>>> df.select(glow.sample_call_summary_stats('genotypes', 'refAllele',
↪'alternateAlleles').alias('stats')).collect()
[Row(stats=[Row(sampleId='NA12878', callRate=1.0, nCalled=3, nUncalled=0, nHomRef=1,
↪ nHet=1, nHomVar=1, nSnp=2, nInsertion=0, nDeletion=1, nTransition=2,␣
↪nTransversion=0, nSpanningDeletion=0, rTiTv=inf, rInsertionDeletion=0.0,␣
↪rHetHomVar=1.0)])]
```

Parameters

- **genotypes** (Union[Column, str]) – An array of genotype structs with calls field
- **refAllele** (Union[Column, str]) – The reference allele
- **alternateAlleles** (Union[Column, str]) – An array of alternate alleles

Return type Column

Returns A struct containing sampleId, callRate, nCalled, nUncalled, nHomRef, nHet, nHomVar, nSnp, nInsertion, nDeletion, nTransition, nTransversion, nSpanningDeletion, rTiTv, rInsertionDeletion, rHetHomVar fields. See *Sample Quality Control*.

glow.**sample_dp_summary_stats**(*genotypes*)

Computes per-sample summary statistics about the depth field in an array of genotype structs.

Added in version 0.3.0.

Examples

```
>>> sites = [
... {'genotypes': [{'sampleId': 'NA12878', 'depth': 1}]},
... {'genotypes': [{'sampleId': 'NA12878', 'depth': 2}]},
... {'genotypes': [{'sampleId': 'NA12878', 'depth': 3}]}]
>>> df = spark.createDataFrame(sites, 'genotypes: array<struct<depth: int,␣
↪sampleId: string>>')
>>> df.select(glow.sample_dp_summary_stats('genotypes').alias('stats')).collect()
[Row(stats=[Row(sampleId='NA12878', mean=2.0, stdDev=1.0, min=1.0, max=3.0)])]
```

Parameters **genotypes** (Union[Column, str]) – An array of genotype structs with depth field

Return type Column

Returns An array of structs where each struct contains mean, stDev, min, and max of the genotype depths for a sample. If sampleId is present in a genotype, it will be propagated to the resulting struct as an extra field.

`glow.`**`sample_gq_summary_stats`**`(`*genotypes*`)`

Computes per-sample summary statistics about the genotype quality field in an array of genotype structs.

Added in version 0.3.0.

Examples

```
>>> sites = [
... Row(genotypes=[Row(sampleId='NA12878', conditionalQuality=1)]),
... Row(genotypes=[Row(sampleId='NA12878', conditionalQuality=2)]),
... Row(genotypes=[Row(sampleId='NA12878', conditionalQuality=3)])]
>>> df = spark.createDataFrame(sites, 'genotypes: array<struct<sampleId: string,␣
↪conditionalQuality: int>>')
>>> df.select(glow.sample_gq_summary_stats('genotypes').alias('stats')).collect()
[Row(stats=[Row(sampleId='NA12878', mean=2.0, stdDev=1.0, min=1.0, max=3.0)])]
```

Parameters **`genotypes`** (`Union`[`Column`, `str`]) – An array of genotype structs with `conditionalQuality` field

Return type `Column`

Returns An array of structs where each struct contains `mean`, `stDev`, `min`, and `max` of the genotype qualities for a sample. If `sampleId` is present in a genotype, it will be propagated to the resulting struct as an extra field.

`glow.`**`subset_struct`**`(`*struct*, **fields*`)`

Selects fields from a struct.

Added in version 0.3.0.

Examples

```
>>> df = spark.createDataFrame([Row(struct=Row(a=1, b=2, c=3))])
>>> df.select(glow.subset_struct('struct', 'a', 'c').alias('struct')).collect()
[Row(struct=Row(a=1, c=3))]
```

Parameters

- **`struct`** (`Union`[`Column`, `str`]) – Struct from which to select fields
- **`fields`** (`str`) – Fields to select

Return type `Column`

Returns A struct containing only the indicated fields

`glow.`**`vector_to_array`**`(`*vector*`)`

Converts a `spark.ml` `Vector` (sparse or dense) to an array of doubles.

Added in version 0.3.0.

Examples

```
>>> from pyspark.ml.linalg import DenseVector, SparseVector
>>> df = spark.createDataFrame([Row(v=SparseVector(3, {0: 1.0, 2: 2.0})),␣
↪Row(v=DenseVector([3.0, 4.0]))])
>>> df.select(glow.vector_to_array('v').alias('arr')).collect()
[Row(arr=[1.0, 0.0, 2.0]), Row(arr=[3.0, 4.0])]
```

Parameters **`vector`** (`Union`[`Column`, `str`]) – Vector to convert

Return type `Column`

Returns An array of doubles

8.3 GloWGR

8.3.1 WGR functions

`class glow.wgr.`**`LogisticRidgeRegression`**(*reduced_block_df*, *label_df*, *sample_blocks*, *cov_df=Empty DataFrame Columns: [] Index: []*, *add_intercept=True*, *alphas=[]*)

The LogisticRidgeRegression class is used to fit logistic ridge regression models against one or more labels optimized over a provided list of ridge alpha parameters. The optimal ridge alpha value is chosen for each label by minimizing the average out of fold log_loss scores.

`fit`()

Fits a logistic regression model, represented by a Spark DataFrame containing coefficients for each of the ridge alpha parameters, for each block in the reduced block matrix, for each label in the target labels, as well as a Spark DataFrame containing the optimal ridge alpha value for each label.

Return type (<class 'pyspark.sql.dataframe.DataFrame'>, <class 'pyspark.sql.dataframe.DataFrame'>)

Returns Two Spark DataFrames, one containing the model resulting from the fitting routine and one containing the results of the cross validation procedure.

`fit_transform`(*response='linear'*)

Fits a logistic ridge regression model, then transforms the matrix using the model.

Parameters **`response`** (`str`) – String specifying the desired output. Can be 'linear' to specify the direct output of the linear WGR model (default) or 'sigmoid' to specify predicted label probabilities.

Return type `DataFrame`

Returns Pandas DataFrame containing prediction y_hat values. The shape and order match labeldf such that the rows are indexed by sample ID and the columns by label. The column types are float64.

`fit_transform_loco`(*response='linear'*, *chromosomes=[]*)

Fits a logistic ridge regression model with a block matrix, then generates predictions for the target labels in the provided label DataFrame by applying the model resulting from the LogisticRidgeRegression fit method to the starting reduced block matrix using a leave-one-chromosome-out (LOCO) approach (this method caches the model and cross-validation DataFrames in the process for better performance).

Parameters

- **response** (str) – String specifying the desired output. Can be 'linear' to specify the direct output of the linear WGR model (default) or 'sigmoid' to specify predicted label probabilities.
- **chromosomes** (List[str]) – List of chromosomes for which to generate a prediction (optional). If not provided, the chromosomes will be inferred from the block matrix.

Return type `DataFrame`

Returns Pandas DataFrame containing prediction y_hat values per chromosome. The rows are indexed by sample ID and chromosome; the columns are indexed by label. The column types are float64. The DataFrame is sorted using chromosome as the primary sort key, and sample ID as the secondary sort key.

`classmethod from_ridge_reduction`(*cls*, *ridge_reduced*, *alphas=[]*)

Initializes an instance of LogsiticRidgeRegression using a RidgeReduction object

Parameters

- **ridge_reduced** (*RidgeReduction*) – A RidgeReduction instance based on which the LogisticRidgeRegression instance must be made
- **alphas** (List[float]) – array_like of alpha values used in the logistic ridge regression (optional).

`reduce_block_matrix`(*response*)

Transforms a starting reduced block matrix by applying a linear model. The form of the output can either be a direct linear transformation (response = "linear") or a linear transformation followed by a sigmoid transformation (response = "sigmoid").

Parameters **response** (str) – String specifying what transformation to apply ("linear" or "sigmoid")

Return type `DataFrame`

Returns Spark DataFrame containing the result of the transformation.

`transform`(*response='linear'*)

Generates GWAS covariates for the target labels in the provided label DataFrame by applying the model resulting from the LogisticRidgeRegression fit method to the starting reduced block matrix.

Parameters **response** (str) – String specifying the desired output. Can be 'linear' to specify the direct output of the linear WGR model (default) or 'sigmoid' to specify predicted label probabilities.

Return type `DataFrame`

Returns Pandas DataFrame containing covariate values. The shape and order match label_df such that the rows are indexed by sample ID and the columns by label. The column types are float64.

`transform_loco`(*response='linear'*, *chromosomes=[]*)

Generates predictions for the target labels in the provided label DataFrame by applying the model resulting from the LogisticRidgeRegression fit method to the starting reduced block matrix using a leave-one-chromosome-out (LOCO) approach (this method caches the model and cross-validation DataFrames in the process for better performance).

Parameters

- **response** (str) – String specifying the desired output. Can be 'linear' to specify the direct output of the linear WGR model (default) or 'sigmoid' to specify predicted label probabilities.

- **chromosomes** (`List[str]`) – List of chromosomes for which to generate a prediction (optional). If not provided, the chromosomes will be inferred from the block matrix.

Return type `DataFrame`

Returns Pandas DataFrame containing prediction y_hat values per chromosome. The rows are indexed by sample ID and chromosome; the columns are indexed by label. The column types are float64. The DataFrame is sorted using chromosome as the primary sort key, and sample ID as the secondary sort key.

`class glow.wgr.`**`RidgeReduction`**(*block_df*, *label_df*, *sample_blocks*, *cov_df=Empty DataFrame Columns: [] Index: []*, *add_intercept=True*, *alphas=[]*, *label_type='detect'*)

The RidgeReduction class is intended to reduce the feature space of an N by M block matrix X to an N by P<<M block matrix. This is done by fitting K ridge models within each block of X on one or more target labels, such that a block with L columns to begin with will be reduced to a block with K columns, where each column is the prediction of one ridge model for one target label.

`fit()`

Fits a ridge reducer model, represented by a Spark DataFrame containing coefficients for each of the ridge alpha parameters, for each block in the starting matrix, for each label in the target labels.

Return type `DataFrame`

Returns Spark DataFrame containing the model resulting from the fitting routine.

`fit_transform()`

Fits a ridge reduction model with a block matrix, then transforms the matrix using the model.

Return type `DataFrame`

Returns Spark DataFrame representing the reduced block matrix

`transform()`

Transforms a starting block matrix to the reduced block matrix, using a reducer model produced by the RidgeReduction fit method.

Return type `DataFrame`

Returns Spark DataFrame representing the reduced block matrix

`class glow.wgr.`**`RidgeRegression`**(*reduced_block_df*, *label_df*, *sample_blocks*, *cov_df=Empty DataFrame Columns: [] Index: []*, *add_intercept=True*, *alphas=[]*)

The RidgeRegression class is used to fit ridge models against one or more labels optimized over a provided list of ridge alpha parameters. It is similar in function to RidgeReduction except that whereas RidgeReduction attempts to reduce a starting matrix X to a block matrix of smaller dimension, RidgeRegression is intended to find an optimal model of the form Y_hat ~ XB, where Y_hat is a matrix of one or more predicted labels and B is a matrix of coefficients. The optimal ridge alpha value is chosen for each label by maximizing the average out of fold r2 score.

`fit()`

Fits a ridge regression model, represented by a Spark DataFrame containing coefficients for each of the ridge alpha parameters, for each block in the starting reduced matrix, for each label in the target labels, as well as a Spark DataFrame containing the optimal ridge alpha value for each label.

Return type (<class 'pyspark.sql.dataframe.DataFrame'>, <class 'pyspark.sql.dataframe.DataFrame'>)

Returns Two Spark DataFrames, one containing the model resulting from the fitting routine and one containing the results of the cross validation procedure.

`fit_transform()`

Fits a ridge regression model, then transforms the matrix using the model.

Return type `DataFrame`

Returns Pandas DataFrame containing prediction y_hat values. The shape and order match labeldf such that the rows are indexed by sample ID and the columns by label. The column types are float64.

`fit_transform_loco`(*chromosomes=[]*)

Fits a ridge regression model and then generates predictions for the target labels in the provided label DataFrame by applying the model resulting from the RidgeRegression fit method to the starting reduced block matrix using a leave-one-chromosome-out (LOCO) approach ((this method caches the model and cross-validation DataFrames in the process for better performance).

Parameters **chromosomes** (List[str]) – List of chromosomes for which to generate a prediction (optional). If not provided, the chromosomes will be inferred from the block matrix.

Return type `DataFrame`

Returns Pandas DataFrame containing offset values (y_hat) per chromosome. The rows are indexed by sample ID and chromosome; the columns are indexed by label. The column types are float64. The DataFrame is sorted using chromosome as the primary sort key, and sample ID as the secondary sort key.

`classmethod from_ridge_reduction`(*cls*, *ridge_reduced*, *alphas=[]*)

Initializes an instance of RidgeRegression using a RidgeReduction object

Parameters

- **ridge_reduced** (*RidgeReduction*) – A RidgeReduction instance based on which the RidgeRegression instance must be made
- **alphas** (List[float]) – array_like of alpha values used in the ridge regression (optional).

`transform`()

Generates predictions for the target labels in the provided label DataFrame by applying the model resulting from the RidgeRegression fit method to the reduced block matrix.

Return type `DataFrame`

Returns Pandas DataFrame containing prediction y_hat values. The shape and order match label_df such that the rows are indexed by sample ID and the columns by label. The column types are float64.

`transform_loco`(*chromosomes=[]*)

Generates predictions for the target labels in the provided label DataFrame by applying the model resulting from the RidgeRegression fit method to the starting reduced block matrix using a leave-one-chromosome-out (LOCO) approach (this method caches the model and cross-validation DataFrames in the process for better performance).

Parameters **chromosomes** (List[str]) – List of chromosomes for which to generate a prediction (optional). If not provided, the chromosomes will be inferred from the block matrix.

Return type `DataFrame`

Returns Pandas DataFrame containing offset values (y_hat) per chromosome. The rows are indexed by sample ID and chromosome; the columns are indexed by label. The column types are float64. The DataFrame is sorted using chromosome as the primary sort key, and sample ID as the secondary sort key.

`glow.wgr.block_variants_and_samples`(*variant_df*, *sample_ids*, *variants_per_block*, *sample_block_count*)

Creates a blocked GT matrix and index mapping from sample blocks to a list of corresponding sample IDs. Uses the same sample-blocking logic as the blocked GT matrix transformer.

Requires that:

- Each variant row has the same number of values
- The number of values per row matches the number of sample IDs

Parameters

- **`variant_df`** (`DataFrame`) – The variant DataFrame
- **`sample_ids`** (`List`[`str`]) – The list of sample ID strings
- **`variants_per_block`** (`int`) – The number of variants per block
- **`sample_block_count`** (`int`) – The number of sample blocks

Return type (<class 'pyspark.sql.dataframe.DataFrame'>, typing.Dict[str, typing.List[str]])

Returns tuple of (blocked GT matrix, index mapping)

`glow.wgr.`**`estimate_loco_offsets`**(*block_df*, *label_df*, *sample_blocks*, *cov_df=Empty DataFrame Columns: [] Index: []*, *add_intercept=True*, *reduction_alphas=[]*, *regression_alphas=[]*, *label_type='detect'*, *chromosomes=[]*)

The one-stop function to generate WGR predictors to be used as offsets in gwas functions. Given the input the function performs the ridge reduction followed by appropriate choice of ridge regression or logistic ridge regression in a loco manner.

Parameters

- **`block_df`** (`DataFrame`) – Spark DataFrame representing the beginning block matrix X
- **`label_df`** (`DataFrame`) – Pandas DataFrame containing the target labels used in fitting the ridge models
- **`sample_blocks`** (`Dict`[`str`, `List`[`str`]]) – Dict containing a mapping of sample_block ID to a list of corresponding sample IDs
- **`cov_df`** (`DataFrame`) – Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).
- **`add_intercept`** (`bool`) – If True, an intercept column (all ones) will be added to the covariates (as the first column)
- **`reduction_alphas`** (`List`[`float`]) – array_like of alpha values used in the ridge reduction (optional). If not provided, the automatically generates alphas for reduction.
- **`regression_alphas`** (`List`[`float`]) – array_like of alpha values used in the ridge or logistic ridge regression (optional). If not provided, the automatically generates alphas for regression.
- **`label_type`** – String to determine type treatment of labels. It can be 'detect' (default), 'binary', or 'quantitative'. On 'detect' the function picks binary or quantitative based on whether label_df is all binary or not, respectively.
- **`chromosomes`** (`List`[`str`]) – List of chromosomes for which to generate offsets (optional). If not provided, the chromosomes will be inferred from the block matrix.

Return type `DataFrame`

Returns Pandas DataFrame containing offset values per chromosome. The rows are indexed by sample ID and chromosome; the columns are indexed by label. The column types are float64. The DataFrame is sorted using chromosome as the primary sort key, and sample ID as the secondary sort key.

`glow.wgr.`**`get_sample_ids`**`(`*data*`)`

Extracts sample IDs from a variant DataFrame, such as one read from PLINK files.

Requires that the sample IDs:

- Are in `genotype.sampleId`
- Are the same across all the variant rows
- Are a list of strings
- Are non-empty
- Are unique

Parameters **`data`** (`DataFrame`) – The variant DataFrame containing sample IDs

Return type `List[str]`

Returns list of sample ID strings

`glow.wgr.`**`reshape_for_gwas`**`(`*spark*, *label_df*`)`

Reshapes a Pandas DataFrame into a Spark DataFrame with a convenient format for Glow's GWAS functions. This function can handle labels that are either per-sample or per-sample and per-contig, like those generated by GloWGR's transform_loco function.

Examples

```
>>> label_df = pd.DataFrame({'label1': [1, 2], 'label2': [3, 4]}, index=['sample1',
↪'sample2'])
>>> reshaped = reshape_for_gwas(spark, label_df)
>>> reshaped.head()
Row(label='label1', values=[1, 2])
```

```
>>> loco_label_df = pd.DataFrame({'label1': [1, 2], 'label2': [3, 4]},
...     index=pd.MultiIndex.from_tuples([('sample1', 'chr1'), ('sample1', 'chr2')]))
>>> reshaped = reshape_for_gwas(spark, loco_label_df)
>>> reshaped.head()
Row(contigName='chr1', label='label1', values=[1])
```

Requires that:

- The input label DataFrame is indexed by sample id or by (sample id, contig name)

Parameters

- **`spark`** (`SparkSession`) – A Spark session
- **`label_df`** (`DataFrame`) – A pandas DataFrame containing labels. The Data Frame should either be indexed by sample id or multi indexed by (sample id, contig name). Each column is interpreted as a label.

Return type `DataFrame`

Returns A Spark DataFrame with a convenient format for Glow regression functions. Each row contains the label name, the contig name if provided in the input DataFrame, and an array containing the label value for each sample.

8.3.2 GWAS functions

`glow.gwas.`**`linear_regression`**(*genotype_df*, *phenotype_df*, *covariate_df=Empty DataFrame Columns: [] Index: []*, *offset_df=Empty DataFrame Columns: [] Index: []*, *contigs=None*, *add_intercept=True*, *values_column='values'*, *dt=<class 'numpy.float64'>*, *verbose_output=False*, *intersect_samples=False*, *genotype_sample_ids=None*)

Uses linear regression to test for association between genotypes and one or more phenotypes. The implementation is a distributed version of the method used in regenie: https://www.biorxiv.org/content/10.1101/2020.06.19.162354v2

Implementation details:

On the driver node, we decompose the covariate matrix into an orthonormal basis and use it to project the covariates out of the phenotype matrix. The orthonormal basis and the phenotype residuals are broadcast as part of a Pandas UDF. In each Spark task, we project the covariates out of a block of genotypes and then compute the regression statistics for each phenotype, taking into account the distinct missingness patterns of each phenotype.

Examples

```
>>> np.random.seed(42)
>>> n_samples, n_phenotypes, n_covariates = (710, 3, 3)
>>> phenotype_df = pd.DataFrame(np.random.random((n_samples, n_phenotypes)),␣
↪columns=['p1', 'p2', 'p3'])
>>> covariate_df = pd.DataFrame(np.random.random((n_samples, n_phenotypes)))
>>> genotype_df = (spark.read.format('vcf').load('test-data/1kg_sample.vcf')
... .select('contigName', 'start', 'genotypes'))
>>> results = glow.gwas.linear_regression(genotype_df, phenotype_df, covariate_df,
... values_column=glow.genotype_states('genotypes'))
>>> results.head()
Row(contigName='1', start=904164, effect=0.0453..., stderror=0.0214..., tvalue=2.
↪114..., pvalue=0.0348..., phenotype='p1')
```

```
>>> phenotype_df = pd.DataFrame(np.random.random((n_samples, n_phenotypes)),␣
↪columns=['p1', 'p2', 'p3'])
>>> covariate_df = pd.DataFrame(np.random.random((n_samples, n_phenotypes)))
>>> genotype_df = (spark.read.format('vcf').load('test-data/1kg_sample.vcf')
... .select('contigName', 'start', 'genotypes'))
>>> contigs = ['1', '2', '3']
>>> offset_index = pd.MultiIndex.from_product([phenotype_df.index, contigs])
>>> offset_df = pd.DataFrame(np.random.random((n_samples * len(contigs), n_
↪phenotypes)),
... index=offset_index, columns=phenotype_df.columns)
>>> results = glow.gwas.linear_regression(genotype_df, phenotype_df, covariate_df,
... offset_df=offset_df, values_column=glow.genotype_states('genotypes'))
```

Parameters

- **`genotype_df`** (`DataFrame`) – Spark DataFrame containing genomic data
- **`phenotype_df`** (`DataFrame`) – Pandas DataFrame containing phenotypic data
- **`covariate_df`** (`DataFrame`) – An optional Pandas DataFrame containing covariates
- **`offset_df`** (`DataFrame`) – An optional Pandas DataFrame containing the phenotype offset, as output by GloWGR's RidgeRegression or Regenie step 1. The actual phenotype used for

linear regression is the mean-centered, residualized and scaled `phenotype_df` minus the appropriate offset. The `offset_df` may have one or two levels of indexing. If one level, the index should be the same as the `phenotype_df`. If two levels, the level 0 index should be the same as the `phenotype_df`, and the level 1 index should be the contig name. The two level index scheme allows for per-contig offsets like LOCO predictions from GloWGR.

- **contigs** (`Optional[List[str]]`) – When using LOCO offsets, this parameter indicates the contigs to analyze. You can use this parameter to limit the size of the broadcasted data, which may be necessary with large sample sizes. If this parameter is omitted, the contigs are inferred from the `offset_df`.
- **add_intercept** (`bool`) – Whether or not to add an intercept column to the covariate DataFrame
- **values_column** (`Union[str, Column]`) – A column name or column expression to test with linear regression. If a column name is provided, `genotype_df` should have a column with this name and a numeric array type. If a column expression is provided, the expression should return a numeric array type.
- **dt** (`type`) – The numpy datatype to use in the linear regression test. Must be `np.float32` or `np.float64`.
- **verbose_output** (`bool`) – Whether or not to generate additional test statistics (n, sum_x, y_transpose_x) to the output DataFrame. These values are derived directly from phenotype_df and genotype_df, and does not reflect any standardization performed as part of the implementation of linear_regression.
- **intersect_samples** (`bool`) – The current implementation of linear regression is optimized for speed, but is not robust to high levels missing phenotype values. Without handling missingness appropriately, pvalues may become inflated due to imputation. When intersect_samples is enabled, samples that do no exist in the phenotype dataframe will be dropped from genotypes, offsets, and covariates prior to regression analysis. Note that if phenotypes in phenotypes_df contain missing values, these samples will not be automatically dropped. The user is responsible for determining their desired levels of missingness and imputation. Drop any rows with missing values from phenotype_df prior to linear_regression to prevent any imputation. If covariates are provided, covariate and phenotype samples will automatically be intersected.
- **genotype_sample_ids** (`Optional[List[str]]`) – Sample ids from genotype_df. i.e. from applying glow.wgr.functions.get_sample_ids(genotype_df) or if include_sample_ids=False was used during the generation genotype_df, then using an externally managed list of sample_ids that correspond to the array of genotype calls.

Return type `DataFrame`

Returns

A Spark DataFrame that contains

- All columns from `genotype_df` except the `values_column` and the `genotypes` column if one exists
- `effect`: The effect size estimate for the genotype
- `stderror`: The estimated standard error of the effect
- `tvalue`: The T statistic
- `pvalue`: P value estimated from a two sided T-test
- `phenotype`: The phenotype name as determined by the column names of `phenotype_df`

- ``n``(int): (verbose_output only) number of samples with non-null phenotype
- ``sum_x``(float): (verbose_output only) sum of genotype inputs
- **``y_transpose_x``(float): (verbose_output only) dot product of phenotype response (missing values encoded a**
  and genotype input, i.e. phenotype value * number of alternate alleles

`glow.gwas.`**`logistic_regression`**(*genotype_df*, *phenotype_df*, *covariate_df=Empty DataFrame Columns: [] Index: []*, *offset_df=Empty DataFrame Columns: [] Index: []*, *correction='approx-firth'*, *pvalue_threshold=0.05*, *contigs=None*, *add_intercept=True*, *values_column='values'*, *dt=<class 'numpy.float64'>*, *intersect_samples=False*, *genotype_sample_ids=None*)

Uses logistic regression to test for association between genotypes and one or more binary phenotypes. This is a distributed version of the method from regenie: https://www.nature.com/articles/s41588-021-00870-7

Implementation details:

On the driver node, we fit a logistic regression model based on the covariates for each phenotype:

$$logit(y) \sim C$$

where y is a phenotype vector and C is the covariate matrix.

We compute the probability predictions $\hat{y}$ and broadcast the residuals $(y - \hat{y})$, γ vectors (where $\gamma = \hat{y} * (1 - \hat{y})$), and $(C^\intercal \gamma C)^{-1}$ matrices. In each task, we then adjust the new genotypes based on the null fit, perform a score test as a fast scan for potentially significant variants, and then test variants with p-values below a threshold using a more selective, more expensive test.

Parameters

- **`genotype_df`** (`DataFrame`) – Spark DataFrame containing genomic data
- **`phenotype_df`** (`DataFrame`) – Pandas DataFrame containing phenotypic data
- **`covariate_df`** (`DataFrame`) – An optional Pandas DataFrame containing covariates
- **`offset_df`** (`DataFrame`) – An optional Pandas DataFrame containing the phenotype offset. This value will be used as an offset in the covariate only and per variant logistic regression models. The `offset_df` may have one or two levels of indexing. If one level, the index should be the same as the `phenotype_df`. If two levels, the level 0 index should be the same as the `phenotype_df`, and the level 1 index should be the contig name. The two level index scheme allows for per-contig offsets like LOCO predictions from GloWGR.
- **`correction`** (`str`) – Which test to use for variants that meet a significance threshold for the score test. Supported methods are `none` and `approx-firth`.
- **`pvalue_threshold`** (`float`) – Variants with a pvalue below this threshold will be tested using the `correction` method.
- **`contigs`** (`Optional[List[str]]`) – When using LOCO offsets, this parameter indicates the contigs to analyze. You can use this parameter to limit the size of the broadcasted data, which may be necessary with large sample sizes. If this parameter is omitted, the contigs are inferred from the `offset_df`.
- **`add_intercept`** (`bool`) – Whether or not to add an intercept column to the covariate DataFrame
- **`values_column`** (`str`) – A column name or column expression to test with linear regression. If a column name is provided, `genotype_df` should have a column with this name and a numeric array type. If a column expression is provided, the expression should return a numeric array type.

- **dt** (type) – The numpy datatype to use in the linear regression test. Must be `np.float32` or `np.float64`.

Return type `DataFrame`

Returns

A Spark DataFrame that contains

- All columns from `genotype_df` except the `values_column` and the `genotypes` column if one exists
- `effect`: The effect size (if approximate Firth correction was applied)
- `stderror`: Standard error of the effect size (if approximate Firth correction was applied)
- `correctionSucceeded`: Whether the correction succeeded (if the correction test method is not `none`). `True` if succeeded, `False` if failed, `null` if correction was not applied.
- `chisq`: The chi squared test statistic according to the score test or the correction method
- `pvalue`: p-value estimated from the test statistic
- `phenotype`: The phenotype name as determined by the column names of `phenotype_df`

8.4 Hail Interoperation Functions

Glow includes functionality to enable interoperation with Hail.

`glow.hail.`**`from_matrix_table`**(*mt*, *include_sample_ids=True*)

Converts a Hail MatrixTable to a Glow DataFrame. The variant fields are derived from the Hail MatrixTable row fields. The sample IDs are derived from the Hail MatrixTable column fields. All other genotype fields are derived from the Hail MatrixTable entry fields.

Requires that the MatrixTable rows contain locus and alleles fields.

Parameters

- **mt** (`MatrixTable`) – The Hail MatrixTable to convert
- **include_sample_ids** (bool) – If true (default), include sample IDs in the Glow DataFrame. Sample names increase the size of each row, both in memory and on storage.

Return type `DataFrame`

Returns Glow DataFrame converted from the MatrixTable.

PYTHON MODULE INDEX

INDEX