Contents

1 Creating datasets using SQL in BigML

It's possible to create new datasets by performing an SQL-style query over a list of input datasets, which are treated as SQL tables. To that end, the POST request JSON should contain the following fields:

- `origin_datasets` A list of objects that describe the input datasets that are going to be used as the input tables of the query. The minimum required information in each object is `id` as the **dataset/id** and a `name` for this dataset to be used as an alias in the `select` clause. You can specify the sample options for each dataset object using the arguments `sample_rate`, `replacement`, `seed`, and `out_of_bag`. Likewise, you can define the range of the dataset by using the `range` option. This is identical to the field used to specify a multi-dataset during merging datasets.
- `sql_query` or `json_query` A string with the SQL query to be executed or a map specifying each of its fields separately respectively. Accepted queries are described in the sections below.
- `sql_output_fields` A list of dictionaries containing some of the properties of the fields generated by the given `sql_query` or `json_query`.

Optionally, you can provide metadata on the fields generated by the query (such as their name, labels or optype) by means of the `sql_output_fields` list, which must therefore have the same length as the number of columns generated by `sql_query` or `json_query`.

1.1 SELECT queries

A SELECT specification can be provided either as a SQL string or as a map possibly containing the following keys:

- **select** A list of strings, each one specifying one of the new fields in the generated dataset. This corresponds to the "selected columns" part of a "SELECT FROM ..." SQL statement, and will use the names in `origin_names` to refer to input datasets as SQL tables. Each table has as one column per dataset field, and its canonical name is the field identifier; but for convenience one can refers to input columns using field names and BigML will translate them automatically to identifiers. For instance, say we have `origin_names: ["d0"]`, i.e. one input dataset with, say, fields `000000`, `000001` and `000002` named **field1**, **field2** and **field3**. One could select only the first field of the first dataset eiher via "SELECT d0.000000" or via "SELECT d0.field1", or using the maps:

  ```
  {"select": ["d0.`000000`"], ...}
  {"select": ["d0.field1"], ...}
  ```

  or the second and first columns of the dataset with, for instance, "SELECT d0.000001, d0.field1", with the map form:

  ```
  {"select": ["d0.`000001`", "d0.field1"], ...}
  ```

  In an SQL query specified as string, you can name the output columns of your query using "AS", for instance: "SELECT d0.field2 AS age" will pick the third column of the dataset and the field of the generated dataset will be named "age". When the query is specified using a JSON dictionary, the corresponding element in the select list will be a pair, with the first element the left hand operator of "AS" and the second element the right hand one. So the previous query would be translated as:

  ```
  {"select": [["d0.field2", "age"]], ...}
  ```

  It is also possible to specify SQL function calls in a select list element, using prefix notation for the operators.

- **distinct** A boolean flag that corresponds to the *distinct* SQL keyword when set to **true**. Defaults to **false**.

- **limit** An integer with the maximum number of rows to select. As in the SQL string "select * limit 3".

- **offset** The offset of the selected rows, as an integer. Corresponds to the *offset* SQL keyword.

- **where** A JSON rendition, as a list, of a SQL *where* clause. The first element in the list is the SQL operand to apply (one of "and", "or",

"count", "avg", "sum", "min", "max", "=", "<>", "!=", ">", ">=", "<", "<=", and "between"), and the rest are its operands (possibly including nested operators). So for instance the SQL clause "d.f0 < 3 and e.f1 = e.f2" is written as

```
["and", ["<", "d.f0", 3], ["=", "e.f1", "e.f2"]]
```

- **having** A translation of a SQL *having* clause using prefix notation, as in `where`.
- **group_by** A list of field identifiers (as in `select`) to perform a SQL *group by* operation.
- **order_by** A list of field identifiers to perform a SQL *order by* operation. Each element in the list can be either a field identifier string, or a pair of a field identifier and either "ASC" or "DESC" to denote ascending or descending ordering. Example: `["t.field1", ["t.field2", "DESC"], ["e.field0" "ASC"]]`.
- **join**, **full_join**, **left_join**, **right_join** Specification of inner, full (outer), left outer and right outer joins. The specification consists of a list starting with the name of the dataset (table) to join on, followed by the operation that one writes in the SQL *ON* specification. Thus, for instance, the SQL string "JOIN foo ON foo.id = bar.id" would be translate to the JSON spec

  ```
  {"join": ["foo", ["=", "foo.id", "bar.id"]]}
  ```

  and likewise for outer joins. Multiple joins using more than two tables are allowed by simply extending the list in the specification with more table/operation pairs (without any nesting); for instance

  ```
  {"select": ["*"],
   "join": ["foo", ["=", "foo.id", "bar.id"],
            "qux", ["=", "qux.id", "bar.id"]],
   "from": ["foo"]}
  ```

  corresponds to the standard SQL query:

  ```
  SELECT * FROM bar
    JOIN foo ON foo.id = bar.id
    JOIN qux ON qux.id = bar.id
  ```

Here's an example of a complicated query combining most of the elements above:

```
{"select": ["f.*", "b.baz", "c.quux", ["b.bla", "bla-bla"], ["now"]],
 "distinct": true,
 "having": ["<", 0, "f.e"],
 "where": ["or", ["and" ["=", "f.a", "bort"]
                        ["!=", "b.baz", "param1"]],
                 ["<", 1, 2, 3],
                 ["in", "f.e", [1, 19, 3]]
                 ["between", "f.e", 10, 20]],
 "limit": 50,
 "group_by": ["f.a"],
 "offset": 10,
 "join": ["draq", ["=", "f.b", "draq.x"]],
 "right_join": ["bock", ["=", "bock.z", "c.e"]],
 "left_join": ["clod", ["=", "f.a", "clod.d"]],
 "order_by": [["b.baz", "desc"], "c.quux", "f.a"]}
```

which would correspond to the SQL query string:

```
SELECT DISTINCT f.*, b.baz, c.quux, b.bla AS bla_bla, now()
INNER JOIN draq ON f.b = draq.x
LEFT JOIN clod c ON f.a = c.d
RIGHT JOIN bock ON bock.z = c.e
WHERE ((f.a = "bort" AND b.baz <> "param1")
       OR (1 < 2 AND 2 < 3)
       OR (f.e in (1, 10, 3))
       OR f.e BETWEEN 10 AND 20)
GROUP BY f.a
HAVING 0 < f.e
ORDER BY b.baz DESC, c.quux, f.a
LIMIT 50
OFFSET 10
```

The user can submit either form as her query. If she uses the latter, as a string, BigML will parse it to a standard map format as the former, discarding non-supported SQL constructs appearing in the query string.

Note that, as is conventional in SQL, we mix freely upper and lowercase keywords in the above examples. BigML should accept both cases, although the recommended style is to not mix them in a single request. Also note that the *JSON keys* in the requests must however all be lower case, as any other JSON key in the interface.

1.1.1 Aggregation functions for pivoting

A common need when constructing ML-datasets is putting together information disseminated in different rows as columns of a transformed dataset, aggregating over a group. To that end one can use `group_by` to specify the field that indicates that the rows in a group belong to the same, single destination instance, and then select values using aggregation functions like the following standard ones:

```
avg sum min max median
stddev variance stddev_pop stddev_samp var_pop var_samp
covar_pop covar_samp row_number collect collect-distinct
group_concat group_concatd
```

In addition to them, we might also need to perform an operation commonly called pivoting, whereby we aggregate the values of a column filtered by the value (category) of a second one. For instance, if we have an employees table

Department	Name	Sex	Salary
1	Jonh	M	100000
1	Peter	M	120000
2	Karla	F	110000
2	Julia	F	50000
2	Karl	M	200000
2	nil	M	2000

we might want to group salaries by department, creating a column with, say, the average salary for female employees and another column with the average salary for male employees:

Department	Avg_salary_F	Avg_salary_M
1	null	110000
2	80000	101000

So we need an SQL function similar to `avg`, but that computes the average of a field in the group (Salary in our example) for a fixed value of another field (Sex) in the group. BigML provides such a function, `cat_avg`, which lets you generate the pivot table as:

```
select Department,
      cat_avg(Salary, Sex, 'Female') as Avg_salary_F,
      cat_avg(Salary, Sex, 'Male') as Avg_salary_M,
from A
group_by Department
```

There are `cat_` variants for each of the common aggreation functions:

```
cat_avg cat_median cat_sum cat_min cat_max
```

returning a double value and taking three arguments (the field to aggregate, the pivoting field and the category to filter). If you know that all the values of the filtering field are integers, you can also use the variants:

```
cat_sum_i cat_min_i cat_max_i
```

that compute an integer instead of a double value (in general, neither the average nor the median of a set of integers is guaranteed to be an integer).

Finally, we provide a per-category aggregation function, `cat_count`, that simply counts instances, so that it needs only two arguments (the name of the categorical field, and the category to count). So, to get this table in our running example:

```
Department  Female_no  Male_no
1           0          2
2           2          2
```

one would run the query

```
select Department,
       cat_count(Sex, 'Female') as Female_no
       cat_count(Sex, 'Male') as Male_no
from A
group_by Department
```

Using these functions as building blocks, we can provide for instance higher-level WhizzML primitives that implement common pivoting use cases.

1.1.2 Properties of the SQL-generated fields

Besides the a query specification, the creation request can include a list of maps with values for some of the properties of the fields generated by the given query. These maps must be provided in a list named `sql_output_fields`, which must contain an entry for each of the generated SQL columns that one wants to tweak. Each entry can specify an output column number (under the key `column`) together with any of the following properties:

- `name` The name for the generated field, overriding any `AS` alias in the query.

- `optype` The optype of the generated field (otherwise BigML will try to infer it). For instance, you can force a column to correspond to a text field instead of a categorical.
- `term_analysis` the parameters for performing text analysis on the generated text field (provided that's the type either inferred or specified in `optype`).
- `item_analysis` the parameters for performing item analysis on the generated items field (provided that's the type specified in `optype`; it must be speciied because `items` is never inferred as an optype).
- `refresh_field_type` output fields directly selected from an input dataset without any transformation inherit the optype, name, preferred flag and, if possible, identifiers of their origin. To request that their optype is recomputed, one can use this boolean flag on a per-field basis or set it globally in the request with the key `refresh_field_types`.
- `refresh_preferred` flag analogous to the previous one but indicating recomputation of the preferred flag. This flag can also be set globally in the request body, and will affect all fields that not override it inside `sql_output_fields`.

Here's an example:

```
{"dataset": {
    "origin_datasets": [
      {"id": "dataset/123299659458fdea3d4584d", "name": "A"}
    ],
    "sql_query": "select `000000` as x, `00000a` as z, `00000c` from A",
    "sql_output_fields": [
      {"column": 0,
       "name": "first, a text",
       "optype": "text",
       "term_analysis": {
          "enabled": true,
          "case_sensitive": true}},
      {"column": 1,
       "refresh_field_type": true,
       "optype": "items",
       "item_analysis": {"separator": ";"}}
    ]
  }
}
```

which will generate 3 fields (as per the SQL query), the first one a text named

"first, a text", an items named "z", and a third field with the same name and optype as the field 00000c in the input dataset.

1.2 Examples

1.2.1 Column concatenation

Say you have 2 datasets and want to join them using the field named "id". A query request would look like this:

```
{"dataset": {
    "origin_datasets": [
      {"id": "dataset/123434569876039485683 28ff", "name": "A"},
      {"id": "dataset/3aad3456987603948568328ff", "name": "B"}
    ],
    "sql_query": "select * from B join A ON A.id = B.id"
  }
}
```

or, using the JSON dictionary form of the query:

```
{"dataset": {
    "origin_datasets": [
      {"id": "dataset/123434569876039485683 28ff", "name": "A"},
      {"id": "dataset/3aad3456987603948568328ff", "name": "B"}
    ],
    "json_query": {
      "select": ["*"],
      "from": ["B"],
      "join": ["A", ["=", "A.id", "B.id"]]
    }
  }
}
```