Configuration File

Configuration files are specified usíng the HOCON format - *Human-Optimized Config Object Notation* (https://github.com/lightbend/config/blob/master/HOCON.md), a simplified JSON dialect.

Subgroup Discovery

Subgroup discovery is the task of finding subgroups of a population which exhibit both distributional unusualness and high generality at the same time. A subgroup consists a set of *items* of the form `<attribute> = <value>` , where attribute is the name of a feature and the value being admissable according to a type restriction.

Note: The greater the number of items is, the more computationally costly subgroup discovery is. Several strategies are used to reduce the number of items. For instance, using numerical values induces a potetially infinite number of items. Binning is used to reduce to a finite number. Reduction ofg the number of instances can also be achieved by using conditions. Given, for instance, features `amount` and `currency` one might impose the condition "amount > 10000 && currency = $" for filtering or even further, use a (derived) compound feature (see below) "amount x currency" to reduce the number of items. For the latter, additionally the condition for the featurs `amount` and `currrency` should be set to `false` since otherwise using the compound feature wouöld boost the number of items rather than to rediuce them. The facilitities offered by conditions and derived features should be used wisely to balance the aim of finding interesting subgroups and the computational costs of doing so.

Input data

Input consists of a list of *instances*. Each instance consists of a list of values that includes one target label. Instances must have the same length. They are either rows of a CSV file or a row in a data base table. The columns must have an (attribute) name, either provided as a header of a CSV file or by the data base scheme. In case that the CSV file does not have a header it is assumed that the list of attributes is that of the feature list under the key `features` (see below).

At present, two input modes are supported

- input from a CSV file
- input from a mySQL

The provider is specified by

```
provider = "csvReader"   // csvReader or mySQLDB
```

If data are read from a file, the file needs to be specified (UNIX file path) as well as the properties of the

CSV file

```
csvReader {
    dataFile = <path>                  // e.g. "abc/def/xyz.csv"
    dataFilesHaveHeader = <boolean>   // Default: true
    separator = <character>           // Default: ','
    quoteCharacter = <character>      // Default: '"'
    escapeCharacter = <character>     // Default: '\\'
}
```

For mySQL the specification is a follows

```
mySQLDB {
    host = <string>         // host name default: localhost
    port = <int>            // port as integer
    database = <string>     // name of the database
    table = <string>        // name of the table
    user = <string>
    password = <string>
}
```

Output

Results will be sored in an output file. The format may either be textual of JSON

```
outputFile = <string>   // attribute only, no extension
outputFormat = txt    // "txt" or "json"
```

The keys `outputFormat` and `outputFile` are optional. Default is `txt` and `<name of the config file>_result`.

Statistics Only

One want to check the frequency of features before running the (potentially time consuming) subgroup mining algorithm. If the optional key

```
statisticsOnly = true
```

is set to true, the frequency of features is computed only.

Subgroup Mining Parameters

The following parameters determine the behaviour of the subgroup mining algorithm. The nomenclature follows: Grosskreutz, H., Rüping, S., & Wrobel, S. (2008). Tight optimistic estimates for fast subgroup discovery. Ecml/Pkdd (1), 5211, 440–456.

```
numberOfBestSubgroups = <integer>       // Default: 15
lengthOfSubgroups = <integer>           // Default: 3
maxNumberOfItems = <integer>            // Default: Int.MaxValue
computeClosureOfSubgroups = <boolean>   // Default: false
refineSubgroups = <boolean>             // Default: false
qualityfunction = <string>              // Default: Piatetsky
minimalQuality = <double>               // Default: 0.0
minGenerality = <double>                // Default: 0.0
minProbability =<double>                // Default: 0.0
```

- `numberOfBestSubgroups` - maximal length of the subgroups
- `maxNumberOfItems` - restricts the number of items. Order criterium is frequeny.
- `computeClosureOfSubgroups` - if true, the closure of the k best subgroups are computed.
- `refineSubgroups` - if true, the refinements of subgroups are computed.
- `qualityfunction` - The quality function used. Supported quality functions are presently
  - Type 2
    - Piatetsky (Default): $n(p - p0)$
    - Binomial: $\sqrt{n}(p - p0)$
  - Type N
    - Split: $n \sum_i (p_i - p_{0_i})^2$
    - Pearson: $n \sum_i (p_i - p_{0_i})^2$
    - Gini: $\frac{n}{N-n} \sum_i (p_i - p_{0_i})^2$

  where n is the size of the subgroup, N the size of the database, p_0 is the class distribution of the database and p that of the respective subgroup. Type 2 implies that there are only two labels, one being the target label.

- `minimalQuality` - Required minimal quality
- `minGenerality` - Generality is the number of occurrences of a subgroup divided by the number of instances

Concurrent Execution

At present there are two options fot parallel execution.

```
parallelExecution {
   numberOfWorkers = <integer>                             // Default: 1
   delimitersForParallelExecutionOfTrees =  [ <integers> ] // Default: []
}
```

- `numberOfWorkers` determines the number of processes used for data preparation.

- `delimitersForParallelExecutionOfTrees` is used for parallel subgroup mining. Delimiters are percentages, i.e. for all delimiters $d, 0 < d < 100$. The tree constructed from all instances is split into several subtrees (with the hope) to improve computation time. Note that the split may substantially increase memory size and the speedup is not necessarily as expected, the reason being that subgroup mining is by no means data parallel.
  The implementation so far runs on a single computer. It might be useful in cases but its benefit may become visible if run on a cluster (not yet implemented).

Features

A list of all *features* of interest needs to be specified

```
features = [ <feature> ]
```

A feature consists of

- an *attribute*, i.e. a string refering to a column of the input data. The string must only use letters a - z, A - Z, digits 0 - 9, and the character '_'. An attribute must start with a letter.
- a type - supported types are `Nominal` and `Numeric`. Values of type `Nominal` are strings, values of type `Numeric` are numbers (integers or Doubles),
- If the type of an feature is `Numeric`, a binning method is required. The intervals generated may be overlapping or not. The binning methods supported are
  - interval binning
  - equal width binning
  - equal frequency binning, and
  - entropy binning.

The following format is accepted

```
<feature> ::=
   {
     attribute = <attribute>,
     typ = <type>,
     binning = <binning> // required if the type is Numeric
     condition = <condition>
   }

<type> ::= "Nominal" | "Numeric"



<binning> ::=
    {
      mode = "Interval",
      intervals = [<double>],
      overlapping = <boolean> // optional, default 'false'
    }
  |
    {
      mode = <mode>,
      bins = <number>,
      overlapping = <boolean>  // optional, default 'false'
    }

<mode> ::= "Equalwidth" | "EqualFrequency" | "Entropy"
```

In case of `<provider>` being `csvReader`, if input data is read from an CSV file that has no header, the feature list is used as a header. Below is an example of a feature list with typical entries

Examples:

```
features = [
   { attribute = "id",
     typ = "Nominal" },
   { attribute = "num",
     typ = "Numeric",
     binning = { mode = "Interval", intervals = [50, 100, 125.6] }
   },
   { attribute = "anotherNum",
     typ = "Numeric",
     binning = {mode = "Entropy", bins = 5}
   }
]
```

Labelling

The list of feature must contain a *target feature* of type `Nominal`. The labels of the target feature are listed under `labels`. Values not being listed are subsumed to a group "default". The target feature is specified by

```
target {
   attribute = <attribute>
   labels = [ <string> ]
}
```

The key `target` must be specified. There must be at least one label.

Time

If instances are ordered by a time attribute, this may be specified by

```
time {
   attribute = <attribute>
   format = <time format>
   start = [ <time> ]
   stop = [ <time> ]
}
```

The values of the timestamp attribute must comply with the data format. The data format is that of http://www.joda.org/joda-time/

The time attribute is optional.

Example

```
timestamp {
   attribute = "transmissiondatehour"
   format = "yyyy-MM-dd HH:mm:ss.0"
   start = "2010-08-04 00:00:00.0"
   end = "2010-08-04 00:00:00.0"
}
```

Derived Features

Derived features are optional.

Compound Features

Compund Features are specified as a list

```
compoundFeatures = [ <compoundFeature> ]

compoundFeature :: =
   {
     group = [ <attribute> ],
     condition = <condition>
   }
```

The attributes must be specified in the list of features.

A compound feature groups a list of features to create a new feature with attribute

`Compound(attr1, ..., attrn)` if `group` = `[attr1, ..., attrn]`.

Given an instance, the value of a compound feature `Compound(attr1, ..., attrn)` consists of the values of the grouped features with attribute `attri` in the order of the groupe attributes.

Prefix Features

Prefix features are specified by

```
prefixFeatures = [ <prefixFeature> ]

prefixFeature ::=
    {
      attribute = <attribute>,
      condition = <condition>,
      prefixes = [ <integer> ]
    }
```

For each prefix n a new feature with the attribute `<attribute>_prefix_n` is generated.

Given an instance, the value of the feature `<attribute>_prefix_n` is the value of the feature with attribute `<attribute>` but reduced to a prefix of length n.

Ranged Features

Ranged features are specified as a list

```
rangedFeatures = [ <rangedFeature> ]

rangedFeature ::=
  {
    attribute = <attribute>,
    condition = <condition>,
    ranges = [ <range> ]
  }

range ::= { lo = <double>, hi = <double> }
```

For each range `{ lo = x, hi = y }`, a new feature with the attribute `<attribute>_range(x,y)` will be generated.

Given an instance, the value of the feature `<attribute>_range(x,y)` if `true` if the value v of the feature with attribute `<attribute>` is in the range, i.e. lo <= x and x < hi.

Aggregation Features

If a time attribute exists, features can aggregated over periods of time. The aggregated features cover the period looking backwards from the actual time.

```
aggregateFeatures = [ <aggregateFeature> | <countFeature> ]

<eaggregateFeature> ::=
    {
      groupBy = <attribute>,
      attribute = <attribute>,
      operator = <aggregateOperator>,
      condition = <condition>,
      minimum = <double>,
      periods = [ <period> ],
      binning = <binning>  // 'Entropy' excluded
    }

    <countFeature> ::=
    {
      groupBy = <attribute>,
      attributes = [ <attribute> ],
      operator = <countOperator>,
      condition = <condition>,
      minimum = <integer>,
      periods = [ <period> ]
    }

<aggregateOperator> ::= sum | max | min | mean
<countOperator> ::= exists | count |
<period> ::= <integer>d | <integer>h | <integer>m | <integer>s | <integer>n
```

where "d" stands for day, "h" for hours, 2m" for minutes, "s" for seconds, and 'n' for number of instances. Further

- All attributes must be specified in the list of features
- The keys `groupBy`, `condition`, `minimum` are optional
- `minimum` defines a lower bound. If the counts or the aggregations are smaller than the bound, no features are generated.

For each period, a new feature attribute is generated of the following format

- if the operator is `exists`

  ```
  Aggregate(<period>).exists(<item> && ... && <item>)
  <item> ::= <attribute> == <value>
  ```

- if the operator is `count`

  ```
  Aggregate(<period>).count(<item> && ... && <item>) == <integer>
  ```

- else

  ```
  Aggregate(<period>).<operator>(<item> && ... && <item>)  == <double>
  ```

where the items comprise all the attributes listed under the keyes `groupBy` and `attributes`.

Example

```
aggregators = [
 { groupBy = id,
    attributes = [],
    operator = count,
    condition = "id == 'a'"
    periods = [10s]
  },
  { groupBy = id,
    attributes = [num],
    operator = exists,
    condition = "id >= 2.0"
    periods = [5m]
  },
  {
    groupBy = id,
    attributes = [ num ],
    operator = sum,
    condition = "id == 'a' && num > 0.0",
    periods = [10h],
    binning = {mode = "Entropy", bins = 5}
  }
]
```

Corresponding attributes are, e.g.,

```
Aggregate(10s).count(id == 'a') == 3
Aggregate(5m).exists(id == 'a' && num == 1.0) == 3
Aggregate(10h).sum(id == 'a') num == [200, 300]
```

Instance Filter

The instance filter is a Boolean expression. Instances that do not satisfy the condition are skipped.

```
  start = <time>
   end = <time>

instanceFilter = <condition>
```

Condition

`<condition>` is Boolean expression as string. Identiers used must be in the list of feature attributes. The notation follows Java conventions, e.g.

```
"(feature1 >= 12.0 || feature1 <= -1.0) && feature2 != NaN"

"num >= 1.3 && x == \"a\" "
```