CSC 643 – Big Data & Web Intelligence

> Team1 <- (Ahmad, Sean)

# *Project 3*

Submission Date: November 25, 2019

## Table of Contents

## Task Ranking

- Member 1: **Ahmad Alshawaf**

  Handled 50% of the project from start to complete

  - Task 1: analyzing data
  - Task 2: solving problem 1
  - Task 3: solving problem 2
  - Task 4: solving problem 3
  - Task 5: organizing report

- Member 2: **Sean Sothey**

  Handled 50% of the project from start to complete

  - Task 1: analyzing data
  - Task 2: solving problem 1
  - Task 3: solving problem 2
  - Task 4: solving problem 3
  - Task 5: organizing report

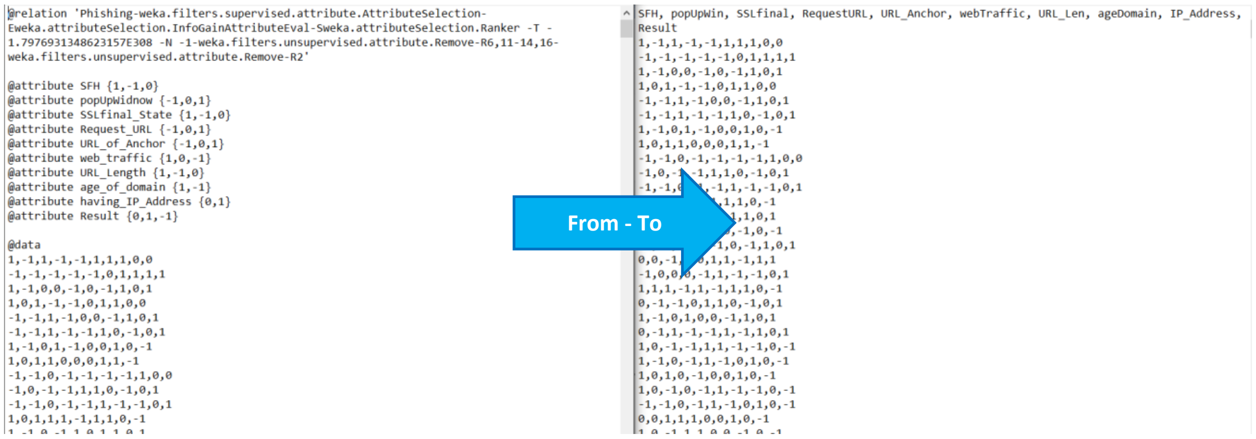
## Introduction

Data science has become an extremely rewarding career choice for people interested in extracting, manipulating, and generating insights out of large volumes of data. To fully leverage the power of data science, scientists often need to obtain skills in databases, statistical programming tools, and data visualizations. Companies surely need data scientists to help them empower their analytics processes, build a numbers-based strategy that will boost their bottom line, and ensure that enormous amounts of data are translated into actionable insights. In this project, we would like to introduce one of the most popular platform tools to analyze the data.

Machine learning and statistics are part of data science. The word learning in machine learning means that the algorithms depend on some data, used as a training set, to fine-tune some model or algorithm parameters. This encompasses many techniques such as regression, naive Bayes or supervised clustering. But not all techniques fit in this category. For instance, unsupervised clustering - a statistical and data science technique - aims at detecting clusters and cluster structures without any a-priori knowledge or training set to help the classification algorithm. A human being is needed to label the clusters found. Some techniques are hybrid, such as semi-supervised classification. Some pattern detection or density estimation techniques fit in this category. RStudio is a powerful tool that can help you do your work better and faster; in technical terms, RStudio is a cross-platform integrated development environment (IDE) for the R statistical language. In this project, we are going to use RStudio, **Sparklyr** and some other libraries and tools to generate an algorithm model that can learn from given training set and produce a significant output.

## Starter Kits

Before, we load the data we need to do a little change to the data file to add the columns name.



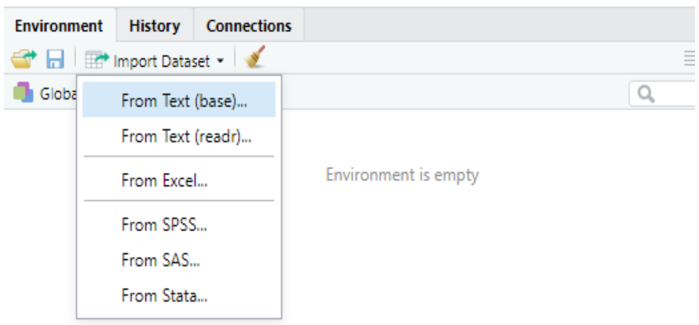
```
@relation 'Phishing-weka.filters.supervised.attribute.AttributeSelection-
Eweka.attributeSelection.InfoGainAttributeEval-Sweka.attributeSelection.Ranker -T -
1.7976931348623157E308 -N -1-weka.filters.unsupervised.attribute.Remove-R6,11-14,16-
weka.filters.unsupervised.attribute.Remove-R2'

@attribute SFH {1,-1,0}
@attribute popUpWidnow {-1,0,1}
@attribute SSLfinal_State {1,-1,0}
@attribute Request_URL {-1,0,1}
@attribute URL_of_Anchor {-1,0,1}
@attribute web_traffic {1,0,-1}
@attribute URL_Length {1,-1,0}
@attribute age_of_domain {1,-1}
@attribute having_IP_Address {0,1}
@attribute Result {0,1,-1}

@data
1,-1,1,-1,-1,1,1,1,0,0
-1,-1,-1,-1,-1,0,1,1,1,1
1,-1,0,0,-1,0,-1,1,0,1
1,0,1,-1,-1,0,1,1,0,0
-1,-1,1,-1,0,0,-1,1,0,1
-1,-1,1,-1,-1,1,0,-1,0,1
1,-1,0,1,-1,0,0,1,0,-1
1,0,1,1,0,0,0,1,1,-1
-1,-1,0,-1,-1,-1,-1,1,0,0
-1,0,-1,-1,1,1,0,-1,0,1
-1,-1,0,-1,-1,1,-1,-1,0,1
1,0,1,1,1,-1,1,1,0,-1
```

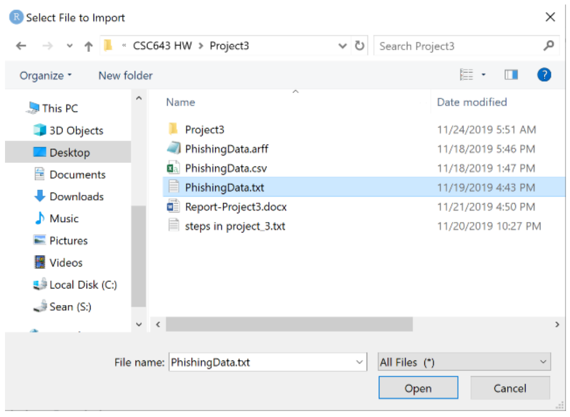
```
SFH, popUpWin, SSLfinal, RequestURL, URL_Anchor, webTraffic, URL_Len, ageDomain, IP_Address,
Result
1,-1,1,-1,-1,1,1,1,0,0
-1,-1,-1,-1,-1,0,1,1,1,1
1,-1,0,0,-1,0,-1,1,0,1
1,0,1,-1,-1,0,1,1,0,0
-1,-1,1,-1,0,0,-1,1,0,1
-1,-1,1,-1,-1,1,0,-1,0,1
1,-1,0,1,-1,0,0,1,0,-1
1,0,1,1,0,0,0,1,1,-1
-1,-1,0,-1,-1,-1,-1,1,0,0
-1,0,-[illegible]-1,1,1,0,-1,0,1
-1,-1,[illegible],-1,1,-1,-1,0,1
[illegible],1,1,0,-1
[illegible]1,1,0,1
[illegible],-1,0,-1
[illegible]1,0,-1,1,0,1
0,0,-1,[illegible],1,1,-1,1,1
-1,0,0,[illegible],-1,1,-1,-1,0,1
1,1,1,-1,1,-1,1,1,0,-1
0,-1,-1,0,1,1,0,-1,0,1
1,-1,0,1,0,0,-1,1,0,1
0,-1,1,-1,-1,1,-1,1,0,1
1,0,-1,-1,1,1,-1,-1,0,-1
1,-1,0,-1,1,-1,0,1,0,-1
1,0,1,0,-1,0,0,1,0,-1
1,0,-1,0,-1,1,-1,-1,0,-1
-1,-1,0,-1,1,-1,0,1,0,-1
0,0,1,1,1,0,0,1,0,-1
```

Then, we load the data, named PhishingData, into RStudio which is the local machine, by typing this command in the RStudio console:

```
PhishingData <- read.csv("C:/Users/seans/Desktop/CSC643 HW/Project3/PhishingData.txt", comment.char="#")
```

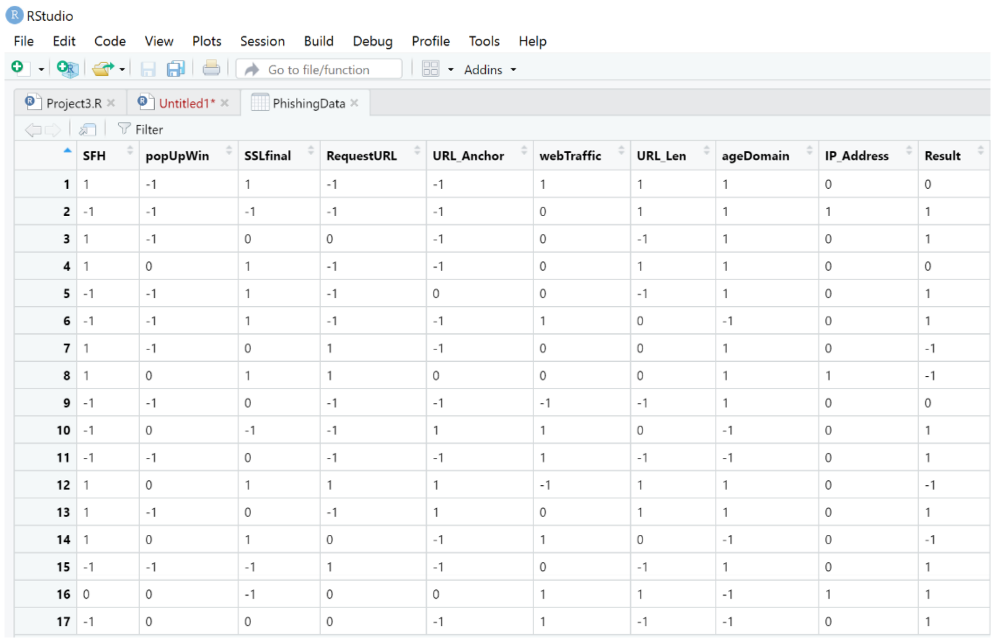
Alternatively, you can do click on **Import Dataset** on the **Environment** layout, then select the file data type you want to load, and so on.





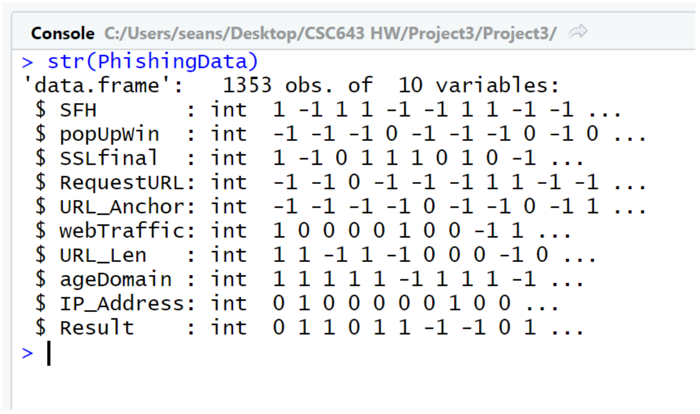
| Name | Date modified |
|---|---|
| Project3 | 11/24/2019 5:51 AM |
| PhishingData.arff | 11/18/2019 5:46 PM |
| PhishingData.csv | 11/18/2019 1:47 PM |
| PhishingData.txt | 11/19/2019 4:43 PM |
| Report-Project3.docx | 11/21/2019 4:50 PM |
| steps in project_3.txt | 11/20/2019 10:27 PM |

File name: PhishingData.txt



| | SFH | popUpWin | SSLfinal | RequestURL | URL_Anchor | webTraffic | URL_Len | ageDomain | IP_Address | Result |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | -1 | 1 | -1 | -1 | 1 | 1 | 1 | 0 | 0 |
| 2 | -1 | -1 | -1 | -1 | -1 | 0 | 1 | 1 | 1 | 1 |
| 3 | 1 | -1 | 0 | 0 | -1 | 0 | -1 | 1 | 0 | 1 |
| 4 | 1 | 0 | 1 | -1 | -1 | 0 | 1 | 1 | 0 | 0 |
| 5 | -1 | -1 | 1 | -1 | 0 | 0 | -1 | 1 | 0 | 1 |
| 6 | -1 | -1 | 1 | -1 | -1 | 1 | 0 | -1 | 0 | 1 |
| 7 | 1 | -1 | 0 | 1 | -1 | 0 | 0 | 1 | 0 | -1 |
| 8 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | -1 |
| 9 | -1 | -1 | 0 | -1 | -1 | -1 | -1 | 1 | 0 | 0 |
| 10 | -1 | 0 | -1 | -1 | 1 | 1 | 0 | -1 | 0 | 1 |
| 11 | -1 | -1 | 0 | -1 | -1 | 1 | -1 | -1 | 0 | 1 |
| 12 | 1 | 0 | 1 | 1 | 1 | -1 | 1 | 1 | 0 | -1 |
| 13 | 1 | -1 | 0 | -1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 14 | 1 | 0 | 1 | 0 | -1 | 1 | 0 | -1 | 0 | -1 |
| 15 | -1 | -1 | -1 | 1 | -1 | 0 | -1 | 1 | 0 | 1 |
| 16 | 0 | 0 | -1 | 0 | 0 | 1 | 1 | -1 | 1 | 1 |
| 17 | -1 | 0 | 0 | 0 | -1 | 1 | -1 | -1 | 0 | 1 |

We can also view the structure of the data set by typing `str(PhishingData)`



```
> str(PhishingData)
'data.frame':	1353 obs. of  10 variables:
 $ SFH       : int  1 -1 1 1 -1 -1 1 1 -1 -1 ...
 $ popUpWin  : int  -1 -1 -1 0 -1 -1 -1 0 -1 0 ...
 $ SSLfinal  : int  1 -1 0 1 1 1 0 1 0 -1 ...
 $ RequestURL: int  -1 -1 0 -1 -1 -1 1 1 -1 -1 ...
 $ URL_Anchor: int  -1 -1 -1 -1 0 -1 -1 0 -1 1 ...
 $ webTraffic: int  1 0 0 0 0 1 0 0 -1 1 ...
 $ URL_Len   : int  1 1 -1 1 -1 0 0 0 -1 0 ...
 $ ageDomain : int  1 1 1 1 1 -1 1 1 1 -1 ...
 $ IP_Address: int  0 1 0 0 0 0 0 1 0 0 ...
 $ Result    : int  0 1 1 0 1 1 -1 -1 0 1 ...
> |
```

And here are some necessary libraries:

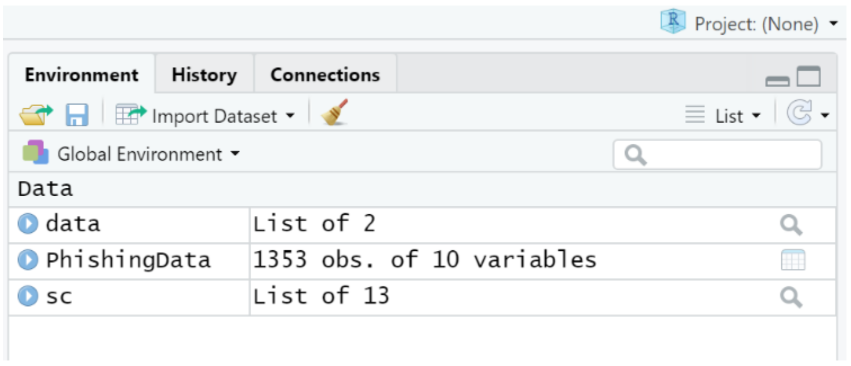
```
library(sparklyr)
library(DBI)
library(dbplyr)
library(BBmisc)
library(party)
```

Now we have the data in Rstudio. Next, we normalize the data format and factor the result by adding a new column called "Result_factor" for labeling . Then load it into Spark Cluster by using spark context (sc).

```
data = normalize(PhishingData, method = "range", range = c(0,2))

data$Result_factor = factor(data$Result, levels = c(0,1,2), labels = c("Phish
y", "Suspicious", "Legitimate"))

sc = spark_connect(master = "local")
```

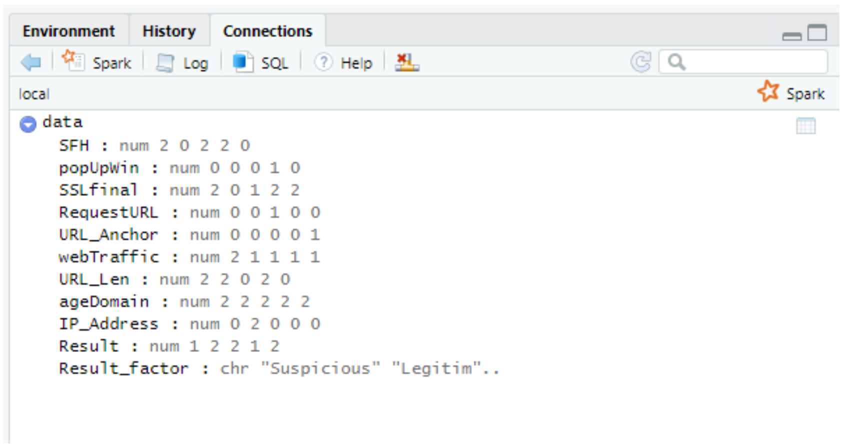
```
data = copy_to(sc, data)
```

Dataset in local



```
data            List of 2
PhishingData    1353 obs. of 10 variables
sc              List of 13
```

Dataset in Spark Cluster



```
data
   SFH : num 2 0 2 2 0
   popUpWin : num 0 0 0 1 0
   SSLfinal : num 2 0 1 2 2
   RequestURL : num 0 0 1 0 0
   URL_Anchor : num 0 0 0 0 1
   webTraffic : num 2 1 1 1 1
   URL_Len : num 2 2 0 2 0
   ageDomain : num 2 2 2 2 2
   IP_Address : num 0 2 0 0 0
   Result : num 1 2 2 1 2
   Result_factor : chr "Suspicious" "Legitim"..
```

## 1. Use the 70%-30% ratio for training and testing

Set the training and testing in ratio 70%-30%

```
partitions = data %>% sdf_random_split(train = 0.7, test = 0.3, seed = 9)
training_set = partitions$train
testing_set = partitions$test
```

Create a training model

```
dt_model = training_set %>%
  ml_decision_tree(Result ~ SFH+popUpWin+SSLfinal+RequestURL+URL_Anchor+
  webTraffic+URL_Len+ageDomain+IP_Address, type = "classification")
```

Define the prediction for both training and testing set to be used for the accuracy

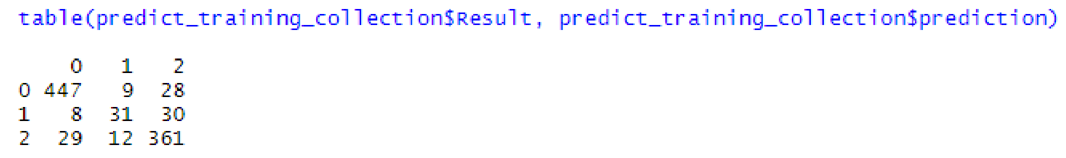
```
predict_training = ml_predict(dt_model, training_set)
predict_testing = ml_predict(dt_model, testing_set)
```

Collect data for both prediction to be used to generate classigication table

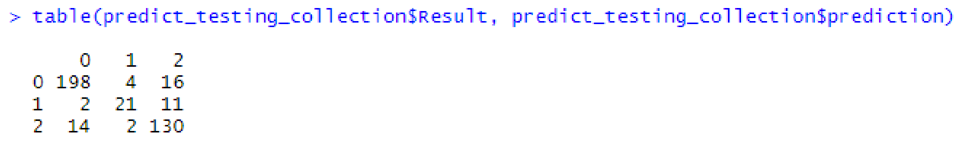
```
predict_training_collection = ml_predict(dt_model, training_set) %>% collect
predict_testing_collection = ml_predict(dt_model, testing_set) %>% collect
```

Create the classification table

```
table(predict_training_collection$Result, predict_training_collection$predict
ion)

table(predict_testing_collection$Result, predict_testing_collection$predictio
n)
```



```
table(predict_training_collection$Result, predict_training_collection$prediction)

     0   1   2
 0 447   9  28
 1   8  31  30
 2  29  12 361
```



```
> table(predict_testing_collection$Result, predict_testing_collection$prediction)

     0   1   2
  0 198   4  16
  1   2  21  11
  2  14   2 130
```

**Training Classification Table**

| | Phishy | Suspicious | Legitimate |
|---|---|---|---|
| phishy | 447 | 9 | 28 |
| Suspicious | 8 | 31 | 30 |
| Legitimate | 29 | 12 | 361 |

**Testing Classification Table**

| | Phishy | Suspicious | Legitimate |
|---|---|---|---|
| phishy | 198 | 4 | 16 |
| Suspicious | 2 | 21 | 11 |
| Legitimate | 14 | 2 | 130 |

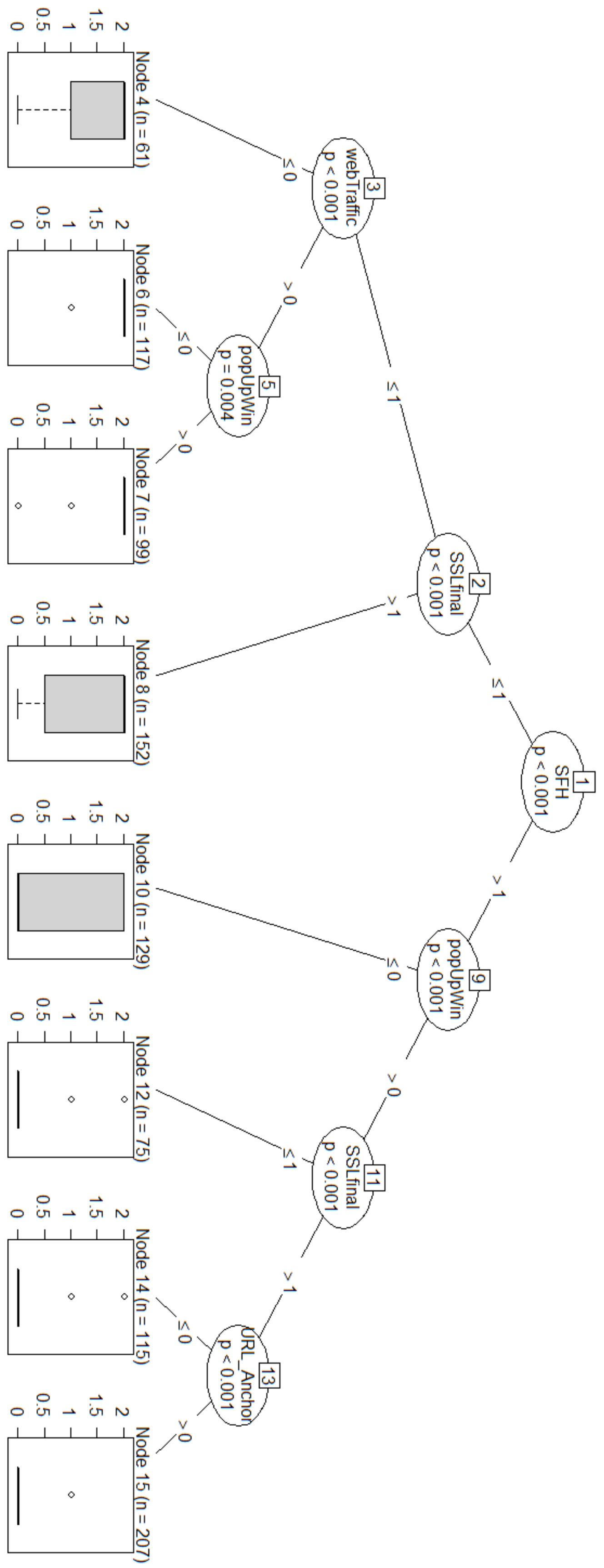
## 2. Generate a decision tree which describes the relationship between the result and the nine categorical attributes

Decision tree of training set

```
training_tree = ctree(Result ~ SFH+popUpWin+SSLfinal+RequestURL+URL_Anchor+we
bTraffic+URL_Len+ageDomain+IP_Address, data = training_set, controls = ctree_
control(mincriterion = 0.99, minsplit = 200))

training_tree
plot(training_tree)
```

```
Conditional inference tree with 8 terminal nodes

Response:  Result
Inputs:  SFH, popUpWin, SSLfinal, RequestURL, URL_Anchor, webTraffic, URL_Len
, ageDomain, IP_Address
Number of observations:  955

1) SFH <= 1; criterion = 1, statistic = 419.884
  2) SSLfinal <= 1; criterion = 1, statistic = 61.742
    3) webTraffic <= 0; criterion = 1, statistic = 17.317
      4)*  weights = 61
    3) webTraffic > 0
      5) popUpWin <= 0; criterion = 0.996, statistic = 12.543
        6)*  weights = 117
      5) popUpWin > 0
        7)*  weights = 99
  2) SSLfinal > 1
    8)*  weights = 152
1) SFH > 1
  9) popUpWin <= 0; criterion = 1, statistic = 93.044
    10)*  weights = 129
  9) popUpWin > 0
    11) SSLfinal <= 1; criterion = 1, statistic = 24.161
      12)*  weights = 75
    11) SSLfinal > 1
      13) URL_Anchor <= 0; criterion = 1, statistic = 23.582
        14)*  weights = 115
      13) URL_Anchor > 0
        15)*  weights = 207
```

***Training Decision Tree***



## 3. Report the classification accuracy of the learner

For training set

```
ml_multiclass_classification_evaluator(predict_training, label_col= 'Result')
[1] 0.8752674
the accuracy of the learner of the training set is 87.5%
```

For testing set

```
ml_multiclass_classification_evaluator(predict_testing, label_col= 'Result')
[1] 0.8756877
the accuracy of the learner of the testing set is 87.5%
```

## Alternative Solution

This is an alternative solution for this project that the computation would be running on your local machine memories instead of Spark Cluster.

-
  ```
  #import data into RStudio
  PhishingData <- read.csv("C:/Users/seans/Desktop/CSC643 HW/Project3/
  PhishingData.txt", comment.char="#")
  ```

-
  ```
  #easy to use it later and not interfere the original data
  data = PhishingData
  ```

-
  ```
  #list all 10 variables information
  str(data)
  ```

-
  ```
  #create a new variable that store the relationship between the
  Result and 9 catagorical attributes
  data$Re_factor = factor(data$Result)
  ```

-
  ```
  #Partision(spliting data)
  #preserve the result every single time; without this, the data slpit
  up will always be different
  set.seed(123)
  ```

-
  ```
  #set data to randomly genterate partision 70-30
  partision = sample(nrow(data), 0.70 * nrow(data))
  ```

-
  ```
  #set training 70
  train = data[partision, ]
  ```

-
  ```
  #set testing 30
  test = data[-partistion, ]
  ```

-
  ```
  #decision tree function for training
  tree_train = ctree(Re_factor~SFH+popUpWin+SSLfinal+RequestURL+
  URL_Anchor+webTraffic+URL_Len+ageDomain+IP_Address, data = train)

  #view the tree and export to pdf(size A3: 8.5 X 22)
  plot(tree_train)
  ```

-
  ```
  #decision tree function for testing
  ```

```
tree_test = ctree(Re_factor~SFH+popUpWin+SSLfinal+RequestURL+
URL_Anchor+webTraffic+URL_Len+ageDomain+IP_Address, data = test)

#view the tree and export to pdf(size A3: 8.5 X 22)
plot(tree_test)
```

-
  ```
  #Prediction
  #create a table of classification for training
  tbl_train_class = table(predict(tree_train), train$Re_factor)

  #view table
  print(tbl_train_class)

  #find the % of training
  sum(diag(tbl_train_class))/sum(tbl_train_class)

  #create a table of classification for testing
  tbl_test_class = table(predict(tree_test), test$Re_factor)

  #view table
  print(tbl_test_class)

  #find the % of testing
  sum(diag(tbl_test_class))/sum(tbl_test_class)
  ```

## Essential Functions Explains

1. `??normalize:`

## Normalizes numeric data to a given scale.

### Description

Currently implemented for numeric vectors, numeric matrices and data.frame. For matrixes one can operate on rows or columns For data.frames, only the numeric columns are touched, all others are left unchanged. For constant vectors / rows / columns most methods fail, special behaviour for this case is implemented.

The method also handles NAs in in `x` and leaves them untouched.

### Usage

```
normalize(x, method = "standardize", range = c(0, 1), margin = 1L,
  on.constant = "quiet")
```

### Arguments

| | |
|---|---|
| `x` | [`numeric`\|`matrix`\|`data.frame`]<br>Input vector. |
| `method` | [`character(1)`]<br>Normalizing method. Available are:<br>“center”: Subtract mean.<br>“scale”: Divide by standard deviation.<br>“standardize”: Center and scale.<br>“range”: Scale to a given range. |
| `range` | [`numeric(2)`]<br>Range for method “range”. The first value represents the replacement for the min value, the second is the substitute for the max value. So it is possible to reverse the order by giving `range = c(1,0)`. Default is `c(0,1)`. |
| `margin` | [`integer(1)`]<br>1 = rows, 2 = cols. Same is in apply Default is 1. |
| `on.constant` | [`character(1)`]<br>How should constant vectors be treated? Only used, of “method != center”, since this methods does not fail for constant vectors. Possible actions are:<br>“quiet”: Depending on the method, treat them quietly:<br>“scale”: No division by standard deviation is done, input values. will be returned untouched.<br>“standardize”: Only the mean is subtracted, no division is done.<br>“range”: All values are mapped to the mean of the given range.<br>“warn”: Same behaviour as “quiet”, but print a warning message.<br>“stop”: Stop with an error. |

### Value

[`numeric` | `matrix` | `data.frame`].

## 2. `?factor:`

# Factors

### Description

The function `factor` is used to encode a vector as a factor (the terms ‘category’ and ‘enumerated type’ are also used for factors). If argument `ordered` is `TRUE`, the factor levels are assumed to be ordered. For compatibility with S there is also a function `ordered`.

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

### Usage

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x), nmax = NA)

ordered(x, ...)

is.factor(x)
is.ordered(x)

as.factor(x)
as.ordered(x)

addNA(x, ifany = FALSE)
```

### Arguments

| | |
|---|---|
| `x` | a vector of data, usually taking a small number of distinct values. |
| `levels` | an optional vector of the unique values (as character strings) that `x` might have taken. The default is the unique set of values taken by as.character`(x)`, sorted into increasing order *of* `x`. Note that this set can be specified as smaller than `sort(unique(x))`. |
| `labels` | *either* an optional character vector of labels for the levels (in the same order as `levels` after removing those in `exclude`), *or* a character string of length 1. Duplicated values in `labels` can be used to map different values of `x` to the same factor level. |
| `exclude` | a vector of values to be excluded when forming the set of levels. This may be factor with the same level set as `x` or should be a `character`. |
| `ordered` | logical flag to determine if the levels should be regarded as ordered (in the order given). |
| `nmax` | an upper bound on the number of levels; see ‘Details’. |
| `...` | (in `ordered(.)`): any of the above, apart from `ordered` itself. |

`ifany` only add an `NA` level if it is used, i.e. if `any(is.na(x))`.

## Details

The type of the vector `x` is not restricted; it only must have an as.character method and be sortable (by order).

Ordered factors differ from factors only in their class, but methods and the model-fitting functions treat the two classes quite differently.

The encoding of the vector happens as follows. First all the values in `exclude` are removed from `levels`. If `x[i]` equals `levels[j]`, then the `i`-th element of the result is `j`. If no match is found for `x[i]` in `levels` (which will happen for excluded values) then the `i`-th element of the result is set to NA.

Normally the 'levels' used as an attribute of the result are the reduced set of levels after removing those in `exclude`, but this can be altered by supplying `labels`. This should either be a set of new labels for the levels, or a character string, in which case the levels are that character string with a sequence number appended.

`factor(x, exclude = NULL)` applied to a factor without NAs is a no-operation unless there are unused levels: in that case, a factor with the reduced level set is returned. If`exclude` is used, since **R** version 3.4.0, excluding non-existing character levels is equivalent to excluding nothing, and when `exclude` is a character vector, that *is* applied to the levels of `x`. Alternatively, `exclude` can be factor with the same level set as `x` and will exclude the levels present in `exclude`.

The codes of a factor may contain NA. For a numeric `x`, set `exclude = NULL` to make NA an extra level (prints as `<NA>`); by default, this is the last level.

If `NA` is a level, the way to set a code to be missing (as opposed to the code of the missing level) is to use is.na on the left-hand-side of an assignment (as in `is.na(f)[i] <- TRUE`; indexing inside `is.na` does not work). Under those circumstances missing values are currently printed as `<NA>`, i.e., identical to entries of level `NA`.

`is.factor` is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Where `levels` is not supplied, unique is called. Since factors typically have quite a small number of levels, for large vectors `x` it is helpful to supply `nmax` as an upper bound on the number of unique values.

## Value

`factor` returns an object of class `"factor"` which has a set of integer codes the length of `x`with a `"levels"` attribute of mode character and unique `(!`anyDuplicated`(.))` entries. If argument `ordered` is true (or `ordered()` is used) the result has class `c("ordered", "factor")`. Undocumentedly for a long time, `factor(x)` loses all attributes`(x)` but `"names"`, and resets `"levels"` and `"class"`.

Applying `factor` to an ordered or unordered factor returns a factor (of the same type) with just the levels which occur: see also [.factor for a more transparent way to achieve this.

`is.factor` returns `TRUE` or `FALSE` depending on whether its argument is of type factor or not. Correspondingly, `is.ordered` returns `TRUE` when its argument is an ordered factor and `FALSE` otherwise.

`as.factor` coerces its argument to a factor. It is an abbreviated (sometimes faster) form of `factor`.

`as.ordered(x)` returns `x` if this is ordered, and `ordered(x)` otherwise.

`addNA` modifies a factor by turning `NA` into an extra level (so that `NA` values are counted in tables, for instance).

`.valid.factor(object)` checks the validity of a factor, currently only `levels(object)`, and returns `TRUE` if it is valid, otherwise a string describing the validity problem. This function is used for validObject`(<factor>)`.

## 3. ? sdf_random_split:

# Partition a Spark Dataframe

### Description

Partition a Spark DataFrame into multiple groups. This routine is useful for splitting a DataFrame into, for example, training and test datasets.

### Usage

```
sdf_random_split(x, ..., weights = NULL,
  seed = sample(.Machine$integer.max, 1))

sdf_partition(x, ..., weights = NULL,
  seed = sample(.Machine$integer.max, 1))
```

### Arguments

| | |
|---|---|
| `x` | An object coercable to a Spark DataFrame. |
| `...` | Named parameters, mapping table names to weights. The weights will be normalized such that they sum to 1. |
| `weights` | An alternate mechanism for supplying weights – when specified, this takes precedence over the `...` arguments. |

| | |
|---|---|
| `seed` | Random seed to use for randomly partitioning the dataset. Set this if you want your partitioning to be reproducible on repeated runs. |

### Details

The sampling weights define the probability that a particular observation will be assigned to a particular partition, not the resulting size of the partition. This implies that partitioning a DataFrame with, for example,

```
sdf_random_split(x, training = 0.5, test = 0.5)
```

is not guaranteed to produce `training` and `test` partitions of equal size.

### Value

An **R** `list` of `tbl_spark`s.

### Transforming Spark DataFrames

The family of functions prefixed with `sdf_` generally access the Scala Spark DataFrame API directly, as opposed to the `dplyr` interface which uses Spark SQL. These functions will 'force' any pending SQL in a `dplyr` pipeline, such that the resulting `tbl_spark` object returned will no longer have the attached 'lazy' SQL operations. Note that the underlying Spark DataFrame *does* execute its operations lazily, so that even though the pending set of operations (currently) are not exposed at the **R** level, these operations will only be executed when you explicitly `collect()` the table.

4. `?ml_decision_tree`

## Spark ML – Decision Trees

### Description

Perform classification and regression using decision trees.

### Usage

```
ml_decision_tree_classifier(x, formula = NULL, max_depth = 5,
  max_bins = 32, min_instances_per_node = 1, min_info_gain = 0,
  impurity = "gini", seed = NULL, thresholds = NULL,
  cache_node_ids = FALSE, checkpoint_interval = 10,
  max_memory_in_mb = 256, features_col = "features",
  label_col = "label", prediction_col = "prediction",
  probability_col = "probability",
  raw_prediction_col = "rawPrediction",
  uid = random_string("decision_tree_classifier_"), ...)

ml_decision_tree(x, formula = NULL, type = c("auto", "regression",
  "classification"), features_col = "features", label_col = "label",
  prediction_col = "prediction", variance_col = NULL,
  probability_col = "probability",
  raw_prediction_col = "rawPrediction", checkpoint_interval = 10L,
  impurity = "auto", max_bins = 32L, max_depth = 5L,
  min_info_gain = 0, min_instances_per_node = 1L, seed = NULL,
  thresholds = NULL, cache_node_ids = FALSE, max_memory_in_mb = 256L,
  uid = random_string("decision_tree_"), response = NULL,
  features = NULL, ...)

ml_decision_tree_regressor(x, formula = NULL, max_depth = 5,
  max_bins = 32, min_instances_per_node = 1, min_info_gain = 0,
  impurity = "variance", seed = NULL, cache_node_ids = FALSE,
  checkpoint_interval = 10, max_memory_in_mb = 256,
  variance_col = NULL, features_col = "features",
  label_col = "label", prediction_col = "prediction",
  uid = random_string("decision_tree_regressor_"), ...)
```

## Arguments

| | |
|---|---|
| `x` | A `spark_connection`, `ml_pipeline`, or a `tbl_spark`. |
| `formula` | Used when `x` is a `tbl_spark`. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see ft_r_formula for details. |
| `max_depth` | Maximum depth of the tree (>= 0); that is, the maximum number of nodes separating any leaves from the root of the tree. |
| `max_bins` | The maximum number of bins used for discretizing continuous features and for choosing how to split on features at each node. More bins give higher granularity. |
| `min_instances_per_node` | Minimum number of instances each child must have after split. |
| `min_info_gain` | Minimum information gain for a split to be considered at a tree node. Should be >= 0, defaults to 0. |
| `impurity` | Criterion used for information gain calculation. Supported: "entropy" and "gini" (default) for classification and "variance" (default) for regression. For `ml_decision_tree`, setting `"auto"` will default to the appropriate criterion based on model type. |
| `seed` | Seed for random numbers. |
| `thresholds` | Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the number of classes, with values > 0 excepting that at most one value may be 0. The class with largest value `p/t` is predicted, where `p` is the original probability of that class and `t` is the class's threshold. |
| `cache_node_ids` | If `FALSE`, the algorithm will pass trees to executors to match instances with nodes. If `TRUE`, the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. Defaults to `FALSE`. |
| `checkpoint_interval` | Set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations, defaults to 10. |
| `max_memory_in_mb` | Maximum memory in MB allocated to histogram aggregation. If too small, then 1 node will be split per iteration, and its aggregates may exceed this size. Defaults to 256. |

| | |
|---|---|
| `features_col` | Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by ft_r_formula. |
| `label_col` | Label column name. The column should be a numeric column. Usually this column is output by ft_r_formula. |
| `prediction_col` | Prediction column name. |
| `probability_col` | Column name for predicted class conditional probabilities. |
| `raw_prediction_col` | Raw prediction (a.k.a. confidence) column name. |
| `uid` | A character string used to uniquely identify the ML estimator. |
| `...` | Optional arguments; see Details. |
| `type` | The type of model to fit. `"regression"` treats the response as a continuous variable, while `"classification"` treats the response as a categorical variable. When `"auto"` is used, the model type is inferred based on the response variable type – if it is a numeric type, then regression is used; classification otherwise. |
| `variance_col` | (Optional) Column name for the biased sample variance of prediction. |
| `response` | (Deprecated) The name of the response column (as a length-one character vector.) |
| `features` | (Deprecated) The name of features (terms) to use for the model fit. |

## Details

When `x` is a `tbl_spark` and `formula` (alternatively, `response` and `features`) is specified, the function returns a `ml_model` object wrapping a `ml_pipeline_model` which contains data pre-processing transformers, the ML predictor, and, for classification models, a post-processing transformer that converts predictions into class labels. For classification, an optional argument `predicted_label_col` (defaults to `"predicted_label"`) can be used to specify the name of the predicted label column. In addition to the fitted `ml_pipeline_model`, `ml_model` objects also contain a `ml_pipeline` object where the ML predictor stage is an estimator ready to be fit against data. This is utilized by ml_save with `type = "pipeline"` to faciliate model refresh workflows.

`ml_decision_tree` is a wrapper around `ml_decision_tree_regressor.tbl_spark`and `ml_decision_tree_classifier.tbl_spark` and calls the appropriate method based on model type.

## Value

The object returned depends on the class of `x`.

- `spark_connection`: When `x` is a `spark_connection`, the function returns an instance of a `ml_estimator` object. The object contains a pointer to a Spark `Predictor` object and can be used to compose `Pipeline` objects.
- `ml_pipeline`: When `x` is a `ml_pipeline`, the function returns a `ml_pipeline` with the predictor appended to the pipeline.

- `tbl_spark`: When `x` is a `tbl_spark`, a predictor is constructed then immediately fit with the input `tbl_spark`, returning a prediction model.
- `tbl_spark`, with `formula`: specified When `formula` is specified, the input `tbl_spark` is first transformed using a `RFormula` transformer before being fit by the predictor. The object returned in this case is a `ml_model` which is a wrapper of a `ml_pipeline_model`.

5. `?ml_predict`

# Spark ML – Transform, fit, and predict methods (ml_ interface)

## Description

Methods for transformation, fit, and prediction. These are mirrors of the corresponding sdf-transform-methods.

## Usage

```
is_ml_transformer(x)

is_ml_estimator(x)

ml_fit(x, dataset, ...)

ml_transform(x, dataset, ...)

ml_fit_and_transform(x, dataset, ...)

ml_predict(x, dataset, ...)

## S3 method for class 'ml_model_classification'
ml_predict(x, dataset,
  probability_prefix = "probability_", ...)
```

## Arguments

| | |
|---|---|
| `x` | A `ml_estimator`, `ml_transformer` (or a list thereof), or `ml_model` object. |
| `dataset` | A `tbl_spark`. |
| `...` | Optional arguments; currently unused. |
| `probability_prefix` | String used to prepend the class probability output columns. |

## Details

These methods are

## Value

When `x` is an estimator, `ml_fit()` returns a transformer whereas `ml_fit_and_transform()` returns a transformed dataset. When `x` is a transformer, `ml_transform()` and `ml_predict()` return a transformed dataset. When `ml_predict()` is called on a `ml_model` object, additional columns (e.g. probabilities in case of classification models) are appended to the transformed output for the user's convenience.

6. `??ctree`

# Conditional Inference Trees

## Description

Recursive partitioning for continuous, censored, ordered, nominal and multivariate response variables in a conditional inference framework.

## Usage

```
ctree(formula, data, subset = NULL, weights = NULL,
      controls = ctree_control(), xtrafo = ptrafo, ytrafo = ptrafo,
      scores = NULL)
```

## Arguments

| | |
|---|---|
| `formula` | a symbolic description of the model to be fit. Note that symbols like `:` and `-` will not work and the tree will make use of all variables listed on the rhs of `formula`. |
| `data` | a data frame containing the variables in the model. |
| `subset` | an optional vector specifying a subset of observations to be used in the fitting process. |
| `weights` | an optional vector of weights to be used in the fitting process. Only non-negative integer valued weights are allowed. |
| `controls` | an object of class TreeControl, which can be obtained using ctree_control. |
| `xtrafo` | a function to be applied to all input variables. By default, the ptrafo function is applied. |
| `ytrafo` | a function to be applied to all response variables. By default, the ptrafofunction is applied. |
| `scores` | an optional named list of scores to be attached to ordered factors. |

## Details

Conditional inference trees estimate a regression relationship by binary recursive partitioning in a conditional inference framework. Roughly, the algorithm works as follows: 1) Test the global null hypothesis of independence between any of the input variables and the response (which may be multivariate as well). Stop if this hypothesis cannot be rejected. Otherwise select the input variable with strongest association to the resonse. This association is measured by a p-value corresponding to a test for the partial null hypothesis of a single input variable and the response. 2) Implement a binary split in the selected input variable. 3) Recursively repeate steps 1) and 2).

The implementation utilizes a unified framework for conditional inference, or permutation tests, developed by Strasser and Weber (1999). The stop criterion in step 1) is either based on multiplicity adjusted p-values (`testtype == "Bonferroni"` or `testtype == "MonteCarlo"` in ctree_control), on the univariate p-values (`testtype == "Univariate"`), or on values of the test statistic (`testtype == "Teststatistic"`). In both cases, the criterion is maximized, i.e., 1 - p-value is used. A split is implemented when the criterion exceeds the value given by `mincriterion` as specified in ctree_control. For example, when `mincriterion = 0.95`, the p-value must be smaller than $0.05$ in order to split this node. This statistical approach ensures that the right sized tree is grown and no form of pruning or cross-validation or whatsoever is needed. The selection of the input variable to split in is based on the univariate p-values avoiding a variable selection bias towards input variables with many possible cutpoints.

Multiplicity-adjusted Monte-Carlo p-values are computed following a "min-p" approach. The univariate p-values based on the limiting distribution (chi-square or normal) are computed for each of the random permutations of the data. This means that one should use a quadratic test statistic when factors are in play (because the evaluation of the corresponding multivariate normal distribution is time-consuming).

By default, the scores for each ordinal factor `x` are `1:length(x)`, this may be changed using `scores = list(x = c(1,5,6))`, for example.

Predictions can be computed using predict or treeresponse. The first function accepts arguments `type = c("response", "node", "prob")` where `type = "response"`returns predicted means, predicted classes or median predicted survival times, `type = "node"` returns terminal node IDs (identical to where) and `type = "prob"` gives more information about the conditional distribution of the response, i.e., class probabilities or predicted Kaplan-Meier curves and is identical to treeresponse. For observations with zero weights, predictions are computed from the fitted tree when `newdata = NULL`.

For a general description of the methodology see Hothorn, Hornik and Zeileis (2006) and Hothorn, Hornik, van de Wiel and Zeileis (2006). Introductions for novices can be found in Strobl et al. (2009) and at http://github.com/christophM/overview-ctrees.git.

## Value

An object of class BinaryTree-class.

## References

Helmut Strasser and Christian Weber (1999). On the asymptotic theory of permutation statistics. *Mathematical Methods of Statistics*, **8**, 220–250.

Torsten Hothorn, Kurt Hornik, Mark A. van de Wiel and Achim Zeileis (2006). A Lego System for Conditional Inference. *The American Statistician*, **60**(3), 257–263.

Torsten Hothorn, Kurt Hornik and Achim Zeileis (2006). Unbiased Recursive Partitioning: A Conditional Inference Framework. *Journal of Computational and Graphical Statistics*, **15**(3), 651–674. Preprint available from http://statmath.wu-wien.ac.at/~zeileis/papers/Hothorn+Hornik+Zeileis-2006.pdf

Carolin Strobl, James Malley and Gerhard Tutz (2009). An Introduction to Recursive Partitioning: Rationale, Application, and Characteristics of Classification and Regression Trees, Bagging, and Random forests. *Psychological Methods*, **14**(4), 323–348.

7. ?ml_multiclass_classification_evaluator

# Spark ML - Evaluators

## Description

A set of functions to calculate performance metrics for prediction models. Also see the Spark ML Documentation https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.evaluation.package

## Usage

```
ml_binary_classification_evaluator(x, label_col = "label",
  raw_prediction_col = "rawPrediction", metric_name = "areaUnderROC",
  uid = random_string("binary_classification_evaluator_"), ...)

ml_binary_classification_eval(x, label_col = "label",
  prediction_col = "prediction", metric_name = "areaUnderROC")

ml_multiclass_classification_evaluator(x, label_col = "label",
  prediction_col = "prediction", metric_name = "f1",
  uid = random_string("multiclass_classification_evaluator_"), ...)

ml_classification_eval(x, label_col = "label",
  prediction_col = "prediction", metric_name = "f1")

ml_regression_evaluator(x, label_col = "label",
  prediction_col = "prediction", metric_name = "rmse",
  uid = random_string("regression_evaluator_"), ...)
```

## Arguments

| | |
|---|---|
| `x` | A `spark_connection` object or a `tbl_spark` containing label and prediction columns. The latter should be the output of `sdf_predict`. |
| `label_col` | Name of column string specifying which column contains the true labels or values. |
| `raw_prediction_col` | Raw prediction (a.k.a. confidence) column name. |
| `metric_name` | The performance metric. See details. |
| `uid` | A character string used to uniquely identify the ML estimator. |
| `...` | Optional arguments; currently unused. |
| `prediction_col` | Name of the column that contains the predicted label or value NOT the scored probability. Column should be of type `Double`. |

## Details

The following metrics are supported

- Binary Classification: `areaUnderROC` (default) or `areaUnderPR` (not available in Spark 2.X.)

- Multiclass Classification: `f1` (default), `precision`, `recall`, `weightedPrecision`, `weightedRecall` or `accuracy`; for Spark 2.X: `f1` (default), `weightedPrecision`, `weightedRecall` or `accuracy`.
- Regression: `rmse` (root mean squared error, default), `mse` (mean squared error), `r2`, or `mae` (mean absolute error.)

`ml_binary_classification_eval()` is an alias for `ml_binary_classification_evaluator()` for backwards compatibility.

`ml_classification_eval()` is an alias for `ml_multiclass_classification_evaluator()` for backwards compatibility.

## Value

The calculated performance metric

## References

- Phishing dataset provided by UCI

https://archive.ics.uci.edu/ml/datasets/Website+Phishing

- Spark Machine Learning Library (MLlib)

https://spark.rstudio.com/mlib/

- rpart

https://www.rdocumentation.org/packages/rpart/versions/4.1-15/topics/rpart