sparklyr

Abdelkader Metales

1/23/2020

Introduction

The programming language R has very powerful tools and functions to do almost every thing we want to do, such as wrangling , visualizing, modeling…etc. However, R such as all the classical languages, requires the whole data to be completely loaded into its memory before doing anything, and this is a big disadvantage when we deal with large data set using less powerful machine, so that any even small data manipulation is time consuming, and may be in some cases the data size can exceed the memory size and R fails even to load the data.

However, there are two widely used engines for this type of data **hadoop** and **spark** which both use a distributed system to partition the data into different storage locations and distribute any computation processes among different machines (computing clusters), or among different CPU's inside a single machine.

Spark is more recent and recognized to be more faster than hadoop (2010). **scala** is its native language, but it can also support **SQL** and **java**. Obviously, if you do not know neither spark nor hadoop it would be obvious to choose spark . However, if you are R user and you do not want to spent time to learn the spark languages (scala, or sql) good news for you is that **sparklyr** package (or sparkR) is R interface for spark from which you can use the most of the R codes and other functions from some packages such as dplyr …etc.

In this paper we will go step by step to learn how to use sparklyr by making use of some examples .

Installing sparklyr

Such as any R package we call the function **install.packages** to install sparklyr, but before that make sure you have **java** installed in your system since the programming language **scala** is run by the java virtual machine.

```
#install.packages("sparklyr")
```

Installing spark

We have deliberately installed sparklyr before spark to provide us with the function **spark_install()** that downloads, installs, and configures the latest version of spark at once.

```
#spark_install()
```

Connecting to spark

Usually, spark is designed to create a clusters using multiple machines either physical machines or virtual machines (in the cloud). However, it can also create a local cluster in your single machine by making use of the CPU's, if exist in this machine, to speed up the data processing.

Wherever the clusters are created (local or in cloud), the data processing functions work in the same way, and the only difference is how to create and interact with these clusters. Since this is the case, then we can get started in our local cluster to learn the most basic things of data science such as importing, analyzing, visualizing data, and perform machine learning models using spark via sparklyr.

To connect to spark in the local mode we use the function **spark_connect** as follows.

```
library(sparklyr)
library(tidyverse)
sc<-spark_connect(master = "local")
```

Importing data

If the data is build-in R we load it to the spark memory using the function **copy_to**.

```
mydata<-copy_to(sc,airquality)
```

Then R can get access to this data by the help of sparklyr, for example we can use the dplyr function **glimpse**.

```
glimpse(mydata)
```

```
## Observations: ??
## Variables: 6
## Database: spark_connection
## $ Ozone   <int> 41, 36, 12, 18, NA, 28, 23, 19, 8, NA, 7, 16, 11, 14, 18, 1...
## $ Solar_R <int> 190, 118, 149, 313, NA, NA, 299, 99, 19, 194, NA, 256, 290,...
## $ Wind    <dbl> 7.4, 8.0, 12.6, 11.5, 14.3, 14.9, 8.6, 13.8, 20.1, 8.6, 6.9...
## $ Temp    <int> 67, 72, 74, 62, 56, 66, 65, 59, 61, 69, 74, 69, 66, 68, 58,...
## $ Month   <int> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,...
## $ Day     <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, ...
```

And if the data is stored anywhere outside R with any different format, then sparklyr provides some functions to import these data. For example to load csv file we use the function **spark_read_csv**, and for json we use **spark_read_json**. To get the list of all the sparklyr functions and their usages click here.

For illustration we will call tha data **creditcards** stored in my machine as follows

```
card<-spark_read_csv(sc,"creditcard.csv")
sdf_dim(card)
```

```
## [1] 284807     31
```

As you see using the same connection **sc** we load two data **mydata** and **card**

if we want to show what is going on in spark we call the function **spark_web()** that lead us to the spark website

```
spark_web(sc)
```

Manipulating data

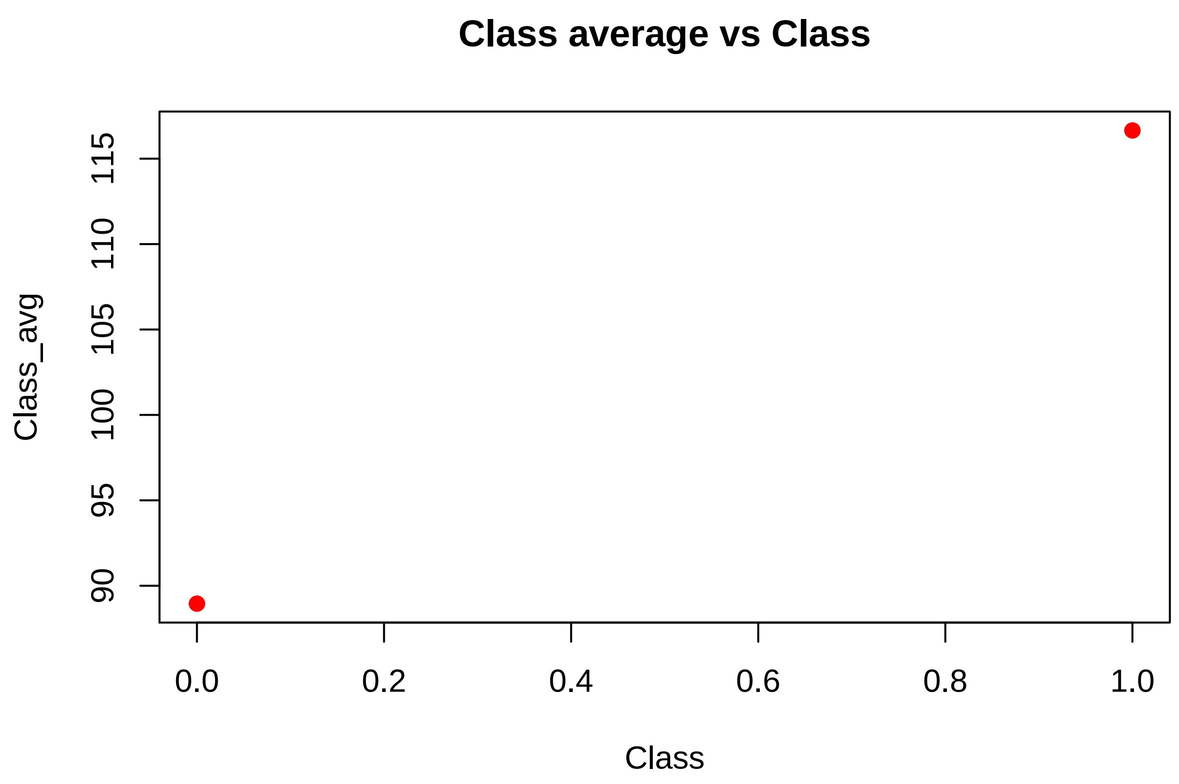
With the help of sparklyr, we can access very easily to the data into spark memory by using the delyr functions. Let's apply some manipulations on the data **card** such as for example filtering the data using the variable **Time** , then we compute the mean of **Amount** for each class label in the variable **Class**.

```
card %>%
  filter(Time <= mean(Time,na.rm = TRUE))%>%
      group_by(Class)%>%
  summarise(Class_avg=mean(Amount,na.rm=TRUE))
```

```
## # Source: spark<?> [?? x 2]
##   Class Class_avg
##   <int>     <dbl>
## 1     0      89.0
## 2     1     117.
```

As you can see now the output is a very small table which can moved from spark memory into R memory for further analysis by making use of the function **collect**. In other words, if you feel with ease in R then each spark output that is small enough to be processed with R add this function at the end of your script before running it to bring this output into R. For example we cannot use the function **plot** to plot the above table, that is why we should fist pull this output into R then apply the function **plot** as follows

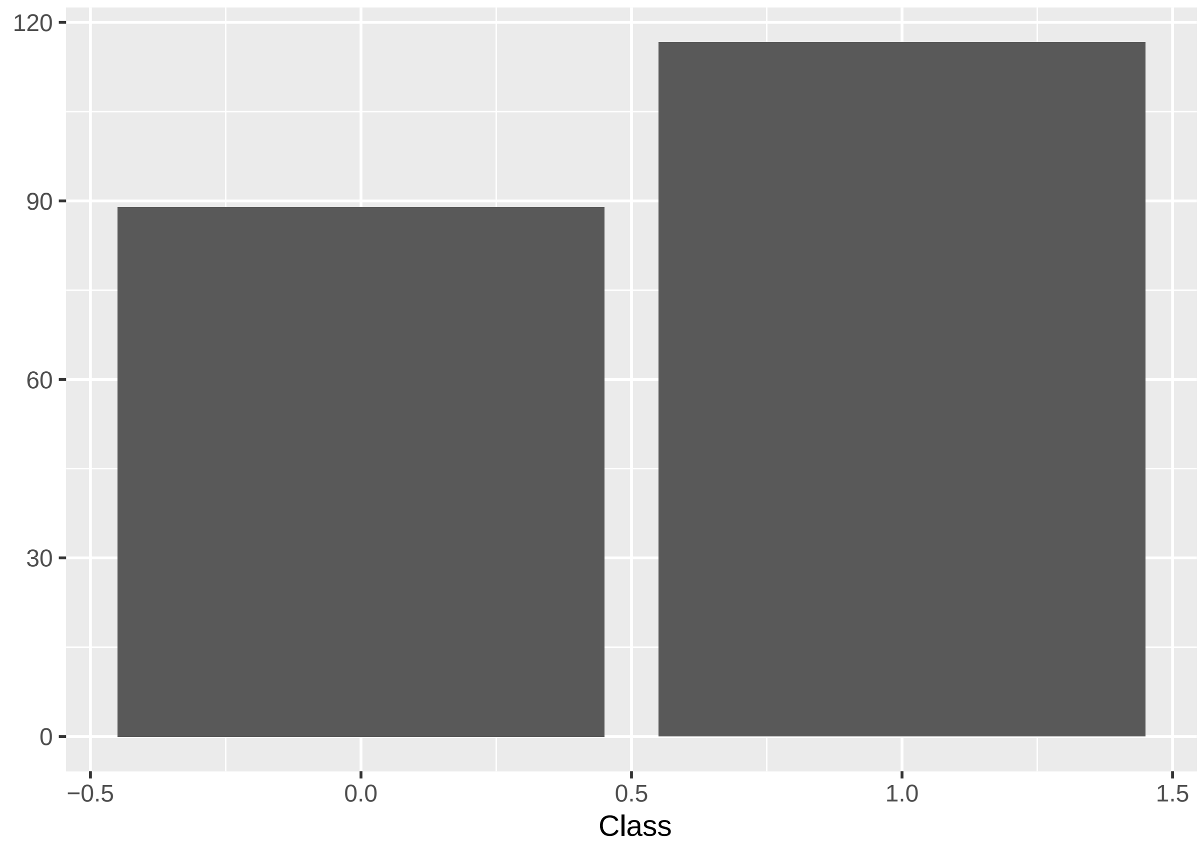
```
card %>%
  filter(Time <= mean(Time,na.rm = TRUE))%>%
      group_by(Class)%>%
  summarise(Class_avg=mean(Amount,na.rm=TRUE))%>%
  collect()%>%
  plot(col="red",pch=19,main = "Class average vs Class")
```



However , we can plot the sparklyr outputs without having to remove them to R memory by using the **dbplot** functions, since most of the functions of this package are supported by sparklyr. Let's for example plot the mean of Amount by Class for cards transaction that have time less than the mean.

```
library(dbplot)
card %>%
  filter(Time <= mean(Time,na.rm = TRUE))%>%
        dbplot_bar(Class,mean(Amount))
```

```
## Warning: Missing values are always removed in SQL.
## Use `mean(x, na.rm = TRUE)` to silence this warning
## This warning is displayed only once per session.
```



As we see the Amount mean of fraudulent cards is higher than that of regular cards.

Disconnecting

each time you finish your work think to disconnect from spark to save your resources as follows.

```
#spark_disconnect(sc)
```

saving data

Sparklyr provides functions to save files directly from spark memory into our directory. For example, to save data in csv file we use spark function **spark_write_csv** (we can save in other type of formats such as **spark_write_parquet**,. . . etc) as follows

```
#spark_write_csv(card,"card.csv")
```

Example of modeling in spark

For machine learning models spark has its own library **MLlib** that has almost every thing we need so that we do not need the library **caret**.

To illustrate how do we perform a machine learning model, we train a logistic regression model to predict the fraudulent cards form the data **card**.

first let's split the data between training set and testing set as follows, and to do this we use the function **sdf_random_split** as follows

```
partitions<-card%>%
  sdf_random_split(training=0.8,test=0.2,seed = 123)
train<-partitions$training
test<-partitions$test
```

Now we will use the set **train** to train our model, and for the model performance we make use of the set **test**.

```
model_in_spark<-train %>%
  ml_logistic_regression(Class~.)
```

we can get the summary of this model by typing its name

```
model_in_spark
```

```
## Formula: Class ~ .
##
## Coefficients:
##   (Intercept)          Time            V1            V2            V3
## -8.305599e+00 -4.074154e-06  1.065118e-01  1.473891e-02 -8.426563e-03
##            V4            V5            V6            V7            V8
##  6.996793e-01  1.380980e-01 -1.217416e-01 -1.205822e-01 -1.700146e-01
##            V9           V10           V11           V12           V13
## -2.734966e-01 -8.277600e-01 -4.476393e-02  7.416858e-02 -2.828732e-01
##           V14           V15           V16           V17           V18
## -5.317753e-01 -1.221061e-01 -2.476344e-01 -1.591295e-03  3.403402e-02
##           V19           V20           V21           V22           V23
##  9.213132e-02 -4.914719e-01  3.863870e-01  6.407714e-01 -1.096256e-01
##           V24           V25           V26           V27           V28
##  1.366914e-01 -5.108841e-02  9.977837e-02 -8.384655e-01 -3.072630e-01
##        Amount
##  1.039041e-03
```

Fortunately, sparklyr also supports the functions of **broom** package so that We can get nicer table using the function **tidy**.

```
library(broom)
tidy(model_in_spark)
```

```
## # A tibble: 31 x 2
##    features    coefficients
##    <chr>              <dbl>
##  1 (Intercept)  -8.31
##  2 Time         -0.00000407
##  3 V1            0.107
##  4 V2            0.0147
##  5 V3           -0.00843
##  6 V4            0.700
##  7 V5            0.138
##  8 V6           -0.122
##  9 V7           -0.121
## 10 V8           -0.170
## # ... with 21 more rows
```

To evaluate the model performance we use the function **ml_evaluate** as follows

```
model_summary<-ml_evaluate(model_in_spark,train)
model_summary
```

```
## BinaryLogisticRegressionSummaryImpl
##  Access the following via `$` or `ml_summary()`.
##  - features_col()
##  - label_col()
##  - predictions()
##  - probability_col()
##  - area_under_roc()
##  - f_measure_by_threshold()
##  - pr()
##  - precision_by_threshold()
##  - recall_by_threshold()
##  - roc()
##  - prediction_col()
##  - accuracy()
##  - f_measure_by_label()
##  - false_positive_rate_by_label()
##  - labels()
##  - precision_by_label()
##  - recall_by_label()
##  - true_positive_rate_by_label()
##  - weighted_f_measure()
##  - weighted_false_positive_rate()
##  - weighted_precision()
##  - weighted_recall()
##  - weighted_true_positive_rate()
```

To extract the metric that we want we use **$**. we can extract for example **the accuracy rate**, the **AUC** or the **roc**

```
model_summary$area_under_roc()
```

```
## [1] 0.9764961
```

```
model_summary$accuracy()
```

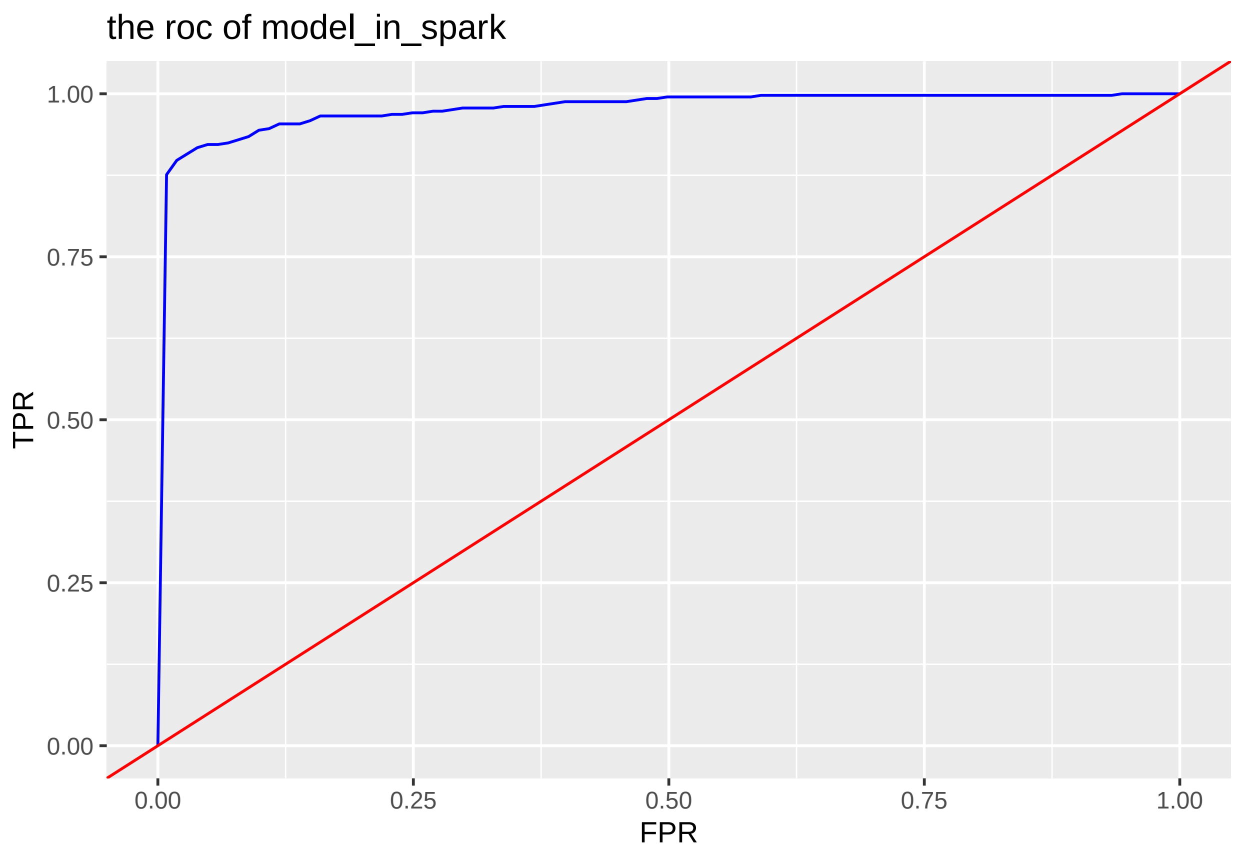
```
## [1] 0.999149
```

```
model_summary$roc()
```

```
## # Source: spark<?> [?? x 2]
##        FPR   TPR
##      <dbl> <dbl>
##  1 0       0
##  2 0.00849 0.876
##  3 0.0185  0.898
##  4 0.0285  0.908
##  5 0.0386  0.917
##  6 0.0487  0.922
##  7 0.0587  0.922
##  8 0.0688  0.925
##  9 0.0788  0.929
## 10 0.0888  0.934
## # ... with more rows
```

we can retrieve this table into R to plot it with ggplot by using the function **collect**

```
model_summary$roc()%>%
collect()%>%
ggplot(aes(FPR,TPR ))+
  geom_line(col="blue")+
  geom_abline(intercept = 0,slope = 1,col="red")+
  ggtitle("the roc of model_in_spark ")
```



High accuracy rate for the training set can be only the result of overfitting problem. the accuracy rate using the testing set is the more reliable one.

```
pred<-ml_evaluate(model_in_spark,test)
pred$accuracy()
```

```
## [1] 0.9994722
```

```
pred$area_under_roc()
```

```
## [1] 0.969233
```

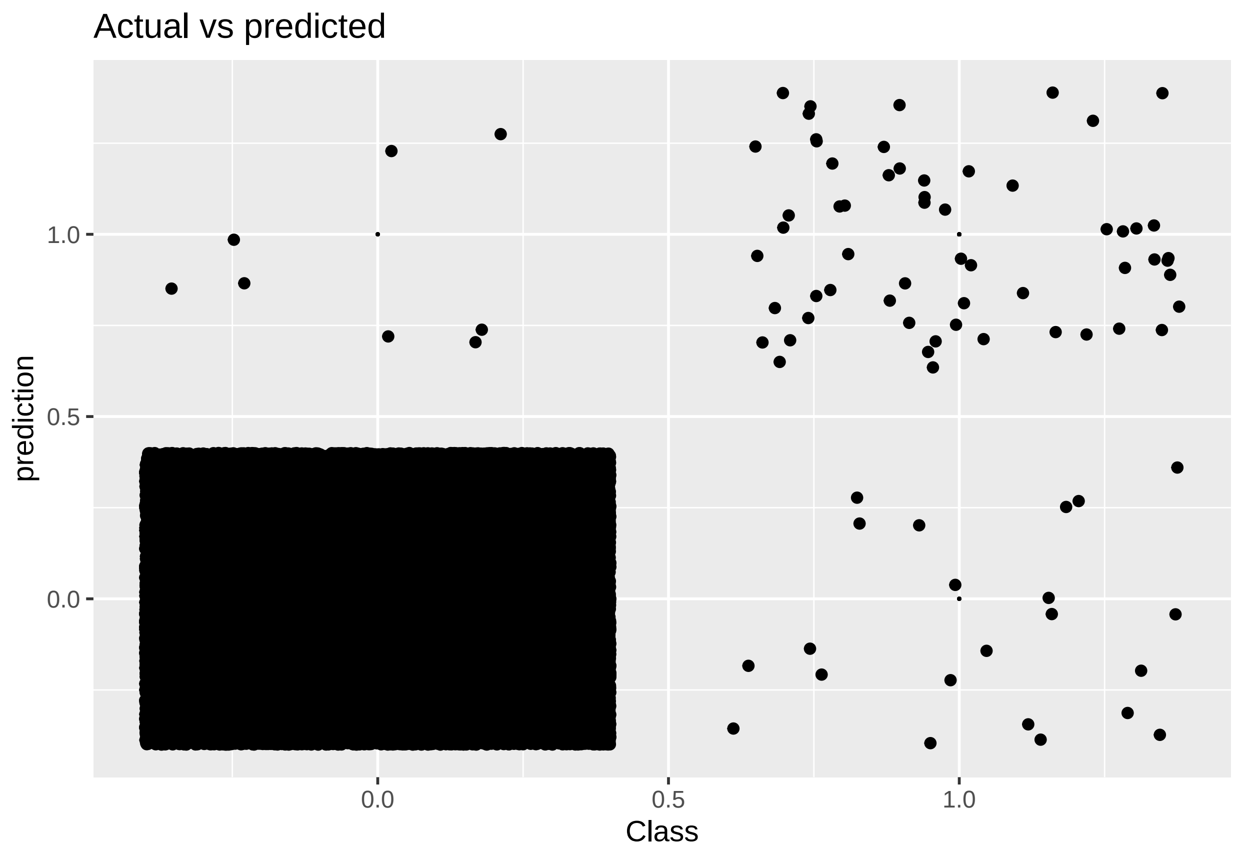
Finally, to get the prediction we use the function **ml_predict**

```
pred<-ml_predict(model_in_spark,test)%>%
select(.,Class,prediction,probability_0,probability_1)
pred
```

```
## # Source: spark<?> [?? x 4]
##    Class prediction probability_0 probability_1
##    <int>      <dbl>         <dbl>         <dbl>
##  1     0          0          1.00      0.000221
##  2     0          0         1.00       0.000441
##  3     0          0         1.00       0.000184
##  4     0          0         1.00       0.000490
##  5     0          0         1.00       0.000199
##  6     0          0         0.999      0.000708
##  7     0          0         1.00       0.000231
##  8     0          0         0.999      0.000640
##  9     0          0         1.00       0.000265
## 10     0          0         0.999      0.000720
## # ... with more rows
```

Here we can also use the function **collect** to plot the results

```
pred%>%
  collect()%>%
  ggplot(aes(Class,prediction ))+
  geom_point(size=0.1)+
  geom_jitter()+
  ggtitle("Actual vs predicted")
```


Actual vs predicted
prediction
Class