logistic regression with sparklyr

Introduction

In one of my previous papers we discussed how to build machine learning model to predict passengers from the **Titanic** data whether they are survived or not using the programming langauge R. In this paper we will use the same data but this time we are using **spark** instead. However, if we do not know the langauges supported by spark such as **scala**, **java**, or **sql**, we can still be able to use spark indirectly via the super package **sparklyr** that translates the most R functions and some other R packages functions such as dplyr,dbplot,corrr,...etc. In addition to the R functions , sparklyr has a bunch of other functions that can be used like any R package. To get the whole list of these functions click here

Since we have a binary classification problem we will train logistic regression model. The most important step of any machine learning model is data preparation, well cleaned and processed data yields in more accurate model.

Data preparation

First we call sparklyr with other packages and we set the connection to spark.

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

Now we can load the data into spark memory. However, if the data is moderately large so that it can be fully loaded into spark memory then we can set the function argument **memory** to be **TRUE** in **spark_read_csv**, but if the data is very large so that its size can exceed the memory size then spark does not stop here like other classical langauges R or python, it has a powrful feature that it can use the data by creating a bridge to it so it sends the queries to the data and retrieve the results without loading it. This can be done by setting the argument **memory** to be FALSE, but since in our case we have small data we keep it as TRUE.

```
mydata<-spark_read_csv(sc,"titanic",path = "train.csv")
```

Sparklyr supports the most deplyr functions, For instance to get a look at this data we can use the dplyr function **glimpse**.

```
glimpse(mydata)
```

```
## Observations: ??
## Variables: 12
## Database: spark_connection
## $ PassengerId <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
## $ Survived    <int> 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0...
## $ Pclass      <int> 3, 1, 3, 1, 3, 3, 1, 3, 3, 2, 3, 1, 3, 3, 3, 2, 3, 2, 3...
## $ Name        <chr> "Braund, Mr. Owen Harris", "Cumings, Mrs. John Bradley ...
## $ Sex         <chr> "male", "female", "female", "female", "male", "male", "...
## $ Age         <dbl> 22, 38, 26, 35, 35, NaN, 54, 2, 27, 14, 4, 58, 20, 39, ...
## $ SibSp       <int> 1, 1, 0, 1, 0, 0, 0, 3, 0, 1, 1, 0, 0, 1, 0, 0, 4, 0, 1...
## $ Parch       <int> 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 1, 0, 0, 5, 0, 0, 1, 0, 0...
## $ Ticket      <chr> "A/5 21171", "PC 17599", "STON/O2. 3101282", "113803", ...
```

```
## $ Fare        <dbl> 7.2500, 71.2833, 7.9250, 53.1000, 8.0500, 8.4583, 51.86...
## $ Cabin       <chr> NA, "C85", NA, "C123", NA, NA, "E46", NA, NA, NA, "G6",...
## $ Embarked    <chr> "S", "C", "S", "S", "S", "Q", "S", "S", "S", "C", "S", ...
```

As we see this data has 12 columns where the variable **survived** will be our target variable and the remaining 11 columns are features. To predict the target variable we should make sure to not use in any machine learning model features that are completly irrelevant, for instance , character feature that has different value for each observation such as **Ticket**, **cabin**, **Name**, and **passengerId** (even it is numeric but in fact it has a character meaning).

Instead of removing these variables we select only the relevant variables, and then we convert the variable **Pclass** to be character. **Note** : we wanted to convert this variable to factor but since the factor type is not supported by sparklyr, we use the character type then when training the model we convert it to dumy variable.

```
newdata<- mydata%>%
  select(c(2,3,5,6,7,8,10,12))%>%
  mutate(Pclass =as.character(Pclass))%>%
  glimpse()
```

```
## Observations: ??
## Variables: 8
## Database: spark_connection
## $ Survived <int> 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1...
## $ Pclass   <chr> "3", "1", "3", "1", "3", "3", "1", "3", "3", "2", "3", "1"...
## $ Sex      <chr> "male", "female", "female", "female", "male", "male", "mal...
## $ Age      <dbl> 22, 38, 26, 35, 35, NaN, 54, 2, 27, 14, 4, 58, 20, 39, 14,...
## $ SibSp    <int> 1, 1, 0, 1, 0, 0, 0, 3, 0, 1, 1, 0, 0, 1, 0, 0, 4, 0, 1, 0...
## $ Parch    <int> 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 1, 0, 0, 5, 0, 0, 1, 0, 0, 0...
## $ Fare     <dbl> 7.2500, 71.2833, 7.9250, 53.1000, 8.0500, 8.4583, 51.8625,...
## $ Embarked <chr> "S", "C", "S", "S", "S", "Q", "S", "S", "S", "C", "S", "S"...
```

When exploring data it is important to check whether you have missing values or not, if yes what is the strategy to use?, removing them from the data or imputing them by applying the appropriate methods. the prefered strategy depends, among others, on the missing values number compared to the data set size, if this number is quite large then the imputation may be the best choice.

```
newdata%>%
  mutate_all(is.na)%>%
  mutate_all(as.numeric)%>%
  summarise_all(sum)
```

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

```
## # Source: spark<?> [?? x 8]
##   Survived Pclass   Sex   Age SibSp Parch  Fare Embarked
##      <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>
## 1        0      0     0   177     0     0     0        2
```

As we see we have large number of missing values. We have 177 missing values from the column **Age** and 2 values from **Embarked**, so we can replace the missing values in Age by the Age median using the sparklyr function **ft_imputer** as follows.

```
newdata<-newdata%>%
  ft_imputer(input_cols = "Age",output_cols="Age",strategy="median")
```

and we can replace those in the column Embarked by the most frequency label. whish is **S**.

```
newdata%>%
  group_by(Embarked)%>%
  count()
```

```
## # Source: spark<?> [?? x 2]
## # Groups: Embarked
##   Embarked     n
##   <chr>    <dbl>
## 1 S          644
## 2 C          168
## 3 <NA>         2
## 4 Q           77
```

```
newdata<-newdata%>%
  na.replace(Embarked="S")
```

To make sure that we do not have anymore missing values, we can re-run the above code.

```
newdata%>%
  mutate_all(is.na)%>%
  mutate_all(as.numeric)%>%
  summarise_all(sum)
```

```
## # Source: spark<?> [?? x 8]
##   Survived Pclass   Sex   Age SibSp Parch  Fare Embarked
##      <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>
## 1        0      0     0     0     0     0     0        0
```

data partition

Now we are ready to split the data between training set and testing set. since we have tow labels we should split the data in a stratified way to preserve label propotions.

```
data_surv<-newdata%>%
  filter(Survived==1)
data_not<-newdata%>%
  filter(Survived==0)
partition_surv<-data_surv%>%
  sdf_random_split(training=0.8,test=0.2,seed = 123)
partition_not<-data_not%>%
  sdf_random_split(training=0.8,test=0.2,seed = 123)

train<-sdf_bind_rows(partition_surv$training,partition_not$training)%>%
    compute("train")
test<-sdf_bind_rows(partition_surv$test,partition_not$test)%>%
    compute("test")
```

Notice that we use the function **compute** this is because sparklyr usually creats a temporarly tables after any transformation so to save the result of any transformation in spark memory we use this function at the end.

The most machine learning models expect the input varaibles to be of numeric type, so we should first convert each character variable to index and then to dumy variable. By its design, sparklyr expects the inputs to be stacked in one column (one vetcor), that is why we call the function **ft_vector_assembler**. At this stage, and before fitting the model, we can also add some other transformations, for instance, we can standardize the inputs to make the model running more faster (especially with models using gradient descent), hence using the function **ft_standard_scaler**, or the function **ft_normalizer** for normalization instead.

```
trained<- train%>%
  ft_string_indexer(input_col = "Sex",output_col="Sex_indexed")%>%
  ft_string_indexer(input_col = "Embarked",output_col="Embarked_indexed")%>%
  ft_string_indexer(input_col = "Pclass",output_col="Pclass_indexed")%>%
  ft_one_hot_encoder_estimator(
    input_cols = c("Pclass_indexed","Sex_indexed","Embarked_indexed"),
    output_cols=c("Pc_encod","Sex_encod","Emb_encod")
  )%>%
  ft_vector_assembler(input_cols = c("Pc_encod","Sex_encod","Age","SibSp",
                                     "Parch","Fare","Emb_encod"),
                      output_col="features")%>%
  ft_standard_scaler(input_col = "features",output_col="scaled",
                     with_mean=TRUE)%>%
  select(Survived,scaled)%>%
  compute("trained")
glimpse(trained)
```

```
## Observations: ??
## Variables: 2
## Database: spark_connection
## $ Survived <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ scaled   <list> [<-1.1143190, 1.7700801, -1.3830544, -1.2168835, 0.455223...
```

Logistic regression model.

For logistic regression we use the function **ml_logistic regression**, and we set the same seed for all the models.

```
model_logistic<-trained%>%
  ml_logistic_regression(features_col = "scaled",
                         label_col="Survived")
```

We can first check the attributes of this model.

```
attributes(model_logistic)
```

```
## $names
##  [1] "uid"                "param_map"          "features_col"
##  [4] "label_col"          "prediction_col"     "num_features"
##  [7] "num_classes"        "probabilitiy_col"   "thresholds"
## [10] "coefficients"       "coefficient_matrix" "intercept"
## [13] "intercept_vector"   "threshold"          "summary"
## [16] ".jobj"
##
## $class
## [1] "ml_logistic_regression_model"
## [2] "ml_probabilistic_classification_model"
## [3] "ml_classification_model"
## [4] "ml_prediction_model"
## [5] "ml_transformer"
## [6] "ml_pipeline_stage"
```

we get the list of the available metrics to measue the model performance as follows.

```
model_logistic$summary
```

```
## BinaryLogisticRegressionTrainingSummaryImpl
```

```
##  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()
```

The most metrics used in practice are **accuracy** and **area_under_roc**, and sometimes we plot the **roc** curve using the **roc** table.

```
model_logistic$summary$accuracy()
```

```
## [1] 0.8176556
```

```
model_logistic$summary$area_under_roc()
```

```
## [1] 0.8662361
```

As we see we have a good rates, but these rates computed from the training set which are usually high, But the challenge for the model is the rates for the testing set. Before including the testing set **test** into the function **ml_evaluate** to get the rates or into the function **ml_predict** to get the predictions, it must follow the same above transformations.

```
tested<-test%>%
  ft_string_indexer(input_col = "Sex",output_col="Sex_indexed")%>%
  ft_string_indexer(input_col = "Embarked",output_col="Embarked_indexed")%>%
  ft_string_indexer(input_col = "Pclass",output_col="Pclass_indexed")%>%
  ft_one_hot_encoder_estimator(
    input_cols = c("Pclass_indexed","Sex_indexed","Embarked_indexed"),
    output_cols=c("Pc_encod","Sex_encod","Emb_encod")
  )%>%
  ft_vector_assembler(input_cols = c("Pc_encod","Sex_encod","Age","SibSp",
                                     "Parch","Fare","Emb_encod"),
                      output_col="features")%>%
  ft_standard_scaler(input_col = "features",output_col="scaled",
                     with_mean=TRUE)%>%
  select(Survived,scaled)%>%
  compute("tested")
```

To evaluate the model using the testing set we use the function **ml_evalaute**, and we can get all the available metrics and others by calling the attributes of the object name in which this output of this function is stored.

```
evaluation<-ml_evaluate(model_logistic,tested)
evaluation$accuracy()
```

```
## [1] 0.76
```

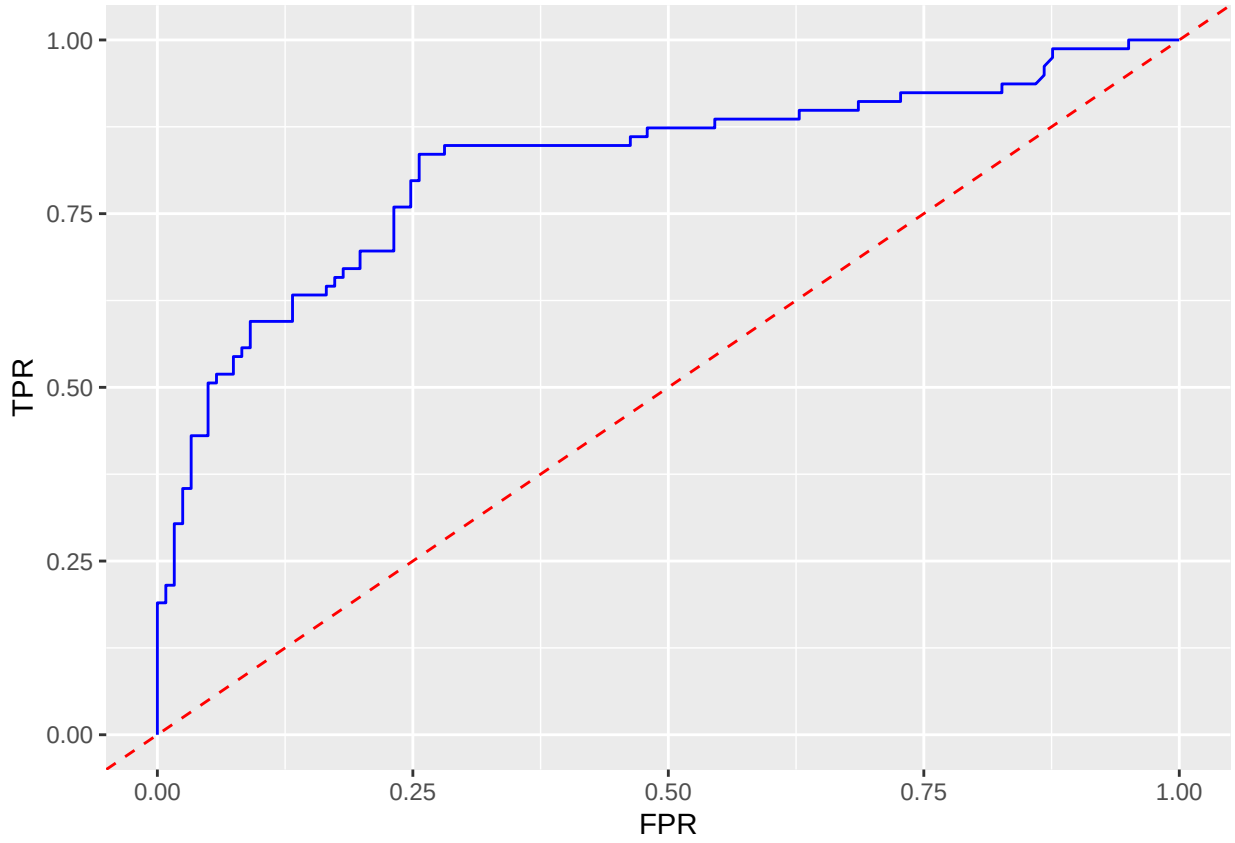
```
evaluation$area_under_roc()
```

```
## [1] 0.8184957
```

As we expected we get lower rates than those for training set.

We can also plot the roc curve after collecting the ouptut into R.

```
evaluation$roc()%>%
  collect()%>%
  ggplot(aes(FPR,TPR))+
  geom_line(col="blue")+
  geom_abline(col="red",lty=2)
```



As a sign of **overfitting** problem, the training rates are more larger than the testing rates which is the case in our data (not very much). to alleviate this problem we can fine-tune the regulariztion parameters of the logistic regression **alpha** and **lambda** using **cross validation**

```
pipline<-ml_pipeline(sc)%>%
  ml_logistic_regression(features_col = "scaled",
                         label_col="Survived")
```

```
cv<-ml_cross_validator(
  sc,
  estimator = pipline,
  estimator_param_maps = list(logistic_regression=list(
    elastic_net_param=c(0,0.1,0.4,0.7),
    reg_param=c(0,0.01,0.1,0.5)
  )),
  evaluator = ml_binary_classification_evaluator(sc,label_col = "Survived"),
  num_folds = 10,
  parallelism = 4,
  seed=1111
)
```

```
cv_model<-ml_fit(cv,trained)
```

Now we get the result as follows

```
ml_validation_metrics(cv_model)%>%
  arrange(-areaUnderROC)
```

```
##    areaUnderROC elastic_net_param_1 reg_param_1
## 1     0.8647517                 0.0        0.10
## 2     0.8632570                 0.1        0.10
## 3     0.8617495                 0.0        0.01
## 4     0.8616667                 0.1        0.01
## 5     0.8603554                 0.4        0.01
## 6     0.8595676                 0.7        0.01
## 7     0.8593622                 0.0        0.00
## 8     0.8593622                 0.1        0.00
## 9     0.8593622                 0.4        0.00
## 10    0.8593622                 0.7        0.00
## 11    0.8585288                 0.4        0.10
## 12    0.8555560                 0.0        0.50
## 13    0.8490092                 0.1        0.50
## 14    0.8459101                 0.7        0.10
## 15    0.7736949                 0.4        0.50
## 16    0.5000000                 0.7        0.50
```

As we see that alpha=0 and lambda=0.1 give the best model. we extract the best model from cv_model to predict the tesing set as follows.

```
pred<-ml_predict(cv_model$best_model,tested)
pred%>%
  sdf_crosstab("Survived","prediction")
```

```
## # Source: spark<?> [?? x 3]
##   Survived_prediction `0.0` `1.0`
##   <chr>                <dbl> <dbl>
## 1 1                       32    47
## 2 0                      108    13
```

```
(108+47)/(108+47+32+13)
```

```
## [1] 0.775
```

from which we get slightly higher accuracy rate 77.5%.

```
spark_disconnect(sc)
```