Local sensitivity hashing model in sparklyr

Introduction

This model is an approximate version for knn model which is difficult to be implemented with large data set. In contrast to knn model that looks for the exact number of nearest neighbours, this model looks for neighbours with high probabilities. Spark provides two methods to find out the approximate neighbours that depend on the data type at hand, **Bucketed random projection** and **Minhash for jaccard distance**.

The first method projects the data in lower dimension hash in which similar hashes indicate that the associated points (or observations) are close to each other. The mathematical basis of this technique is the following formula.

$$h^{x,b}(\vec{v}) = \lfloor \frac{\vec{v}.\vec{x}}{w} \rfloor$$

Where h is the hashing function, $\vec{v}$ is the feature vector, x is standard normal vector that has the same length, and w is the bin width of the hashing bins, and the symbol $\lfloor\rfloor$ to coerce the result to be integer value. The idea is simple, we take the dot product of each feature vector with noisy vector, then the resulted projections (which are random) will be grouped into buckets, these buckets are supposed to include similar points. This process can be repeated many times with different noisy vector at each time to fine the similarity.

For more detail click here

First we set the connection to spark.

```r
library(sparklyr)
library(tidyverse)
sc<-spark_connect(master="local")
mydata<-spark_read_csv(sc,"titanic",path = "train.csv")
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", ...
```

Then we remove some varaibles that we think they are useless.

```
newdata<- mydata%>%
  select(c(2,3,5,6,7,8,10,12))%>%
  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   <int> 3, 1, 3, 1, 3, 3, 1, 3, 3, 2, 3, 1, 3, 3, 3, 2, 3, 2, 3, 3...
$ 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"...
```

May be the first thing we do in explaratory analysis is to check the missing values.

```
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   177     0     0     0        2
```

Since we have a large number of missing values it would be better to imput thes values rather than removing them. For the numeric variable **Age** we replace them by the median using the sparklyr function **ft_imputer**, and for categorical variable **Embarked** we use the most frequantly label which is here **S** port.

```
newdata<-newdata%>%
  ft_imputer(input_cols = "Age",output_cols="Age",strategy="median")%>%
  na.replace(Embarked="S")
```

Now we split the data between training and testing set , then we add an id column to ach set in order to uniquelly identify each row.

```
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)
id<-sdf_len(sc,sdf_nrow(train))
train<-sdf_bind_cols(id,train)%>%
  compute("train")
test<-sdf_bind_rows(partition_surv$test,partition_not$test)
id<-sdf_len(sc,sdf_nrow(test))
test<-sdf_bind_cols(id,test)%>%
  compute("test")
```

Before fitting any model the data must be processed in a way that can be consumed by the model. For our model, such as the most machine learning models, requires numeric features, we convert thus categorical variables to integers using the function **ft__strin__indexer**, after that we convert them to dumy variables using the function **ft__one hot__encoder__estimator**, because the last function expects the inputs to be numeric.

For models build in sparklyr, the inputs variables should be stacked into one column vector on each other, this can be easily done by using the function **ft__vector__assembler**. However, this step does not prevent us to apply some other transformation even the features are in one column. For instance, to run efficiently our model we can transform the variables such that they have the same scale, to do so we can either use standardization (sa we do here) or normalization method.

Now it is a good practice to save this preocessed set into the spark memory under an object name using the function **compute**

```
trained<- train%>%
  ft_string_indexer(input_col = "Sex",output_col="Sex_indexed")%>%
  ft_string_indexer(input_col = "Embarked",output_col="Embarked_indexed")%>%
  ft_one_hot_encoder_estimator(
    input_cols = c("Pclass","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(id,Survived,scaled)%>%
  compute("trained")
```

The same transformations above will be applied to the testing set **test**

```
tested<-test%>%
  ft_string_indexer(input_col = "Sex",output_col="Sex_indexed")%>%
  ft_string_indexer(input_col = "Embarked",output_col="Embarked_indexed")%>%
  ft_one_hot_encoder_estimator(
    input_cols = c("Pclass","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(id,Survived,scaled)%>%
  compute("tested")
```

Now we are ready to project the data on the lower dimension hash using the function

ft__bucketed__random__projection__lsh

```
lsh_vector<-ft_bucketed_random_projection_lsh(sc,
                                              input_col = "scaled",
                                              output_col = "hash",
                                              bucket_length = 2,
                                              num_hash_tables = 5,
                                              seed=111)
```

Now we fit this model to the training data **trained**

```
model_lsh<-ml_fit(lsh_vector,trained)
```

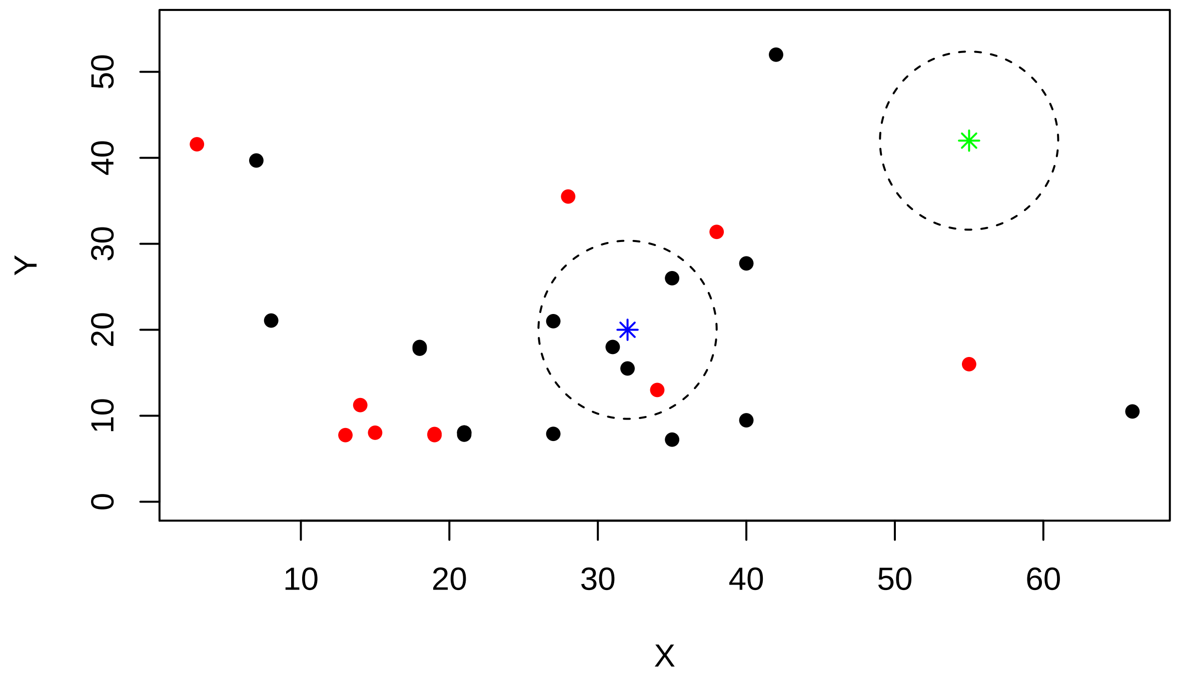
Prediction

At the prediction stage this model of classification give us to alternatives for how we define the nearest neighbours:

- define a threshold value from which we decide if two observations are considered as nearest neighbours or not, small value leads to take small number of neighbours. in sparklyr we can achive that using the function **ml__approx__similarity__join** and we specify the the threshold value for the minimum distance. the distance used by this function is the classical euclidien distance.
- prespecify the number of the nearest neighbours without regardeless of the distance between observations. This second alternative can be achieved using **ml__approx__nearest__neighbors**.

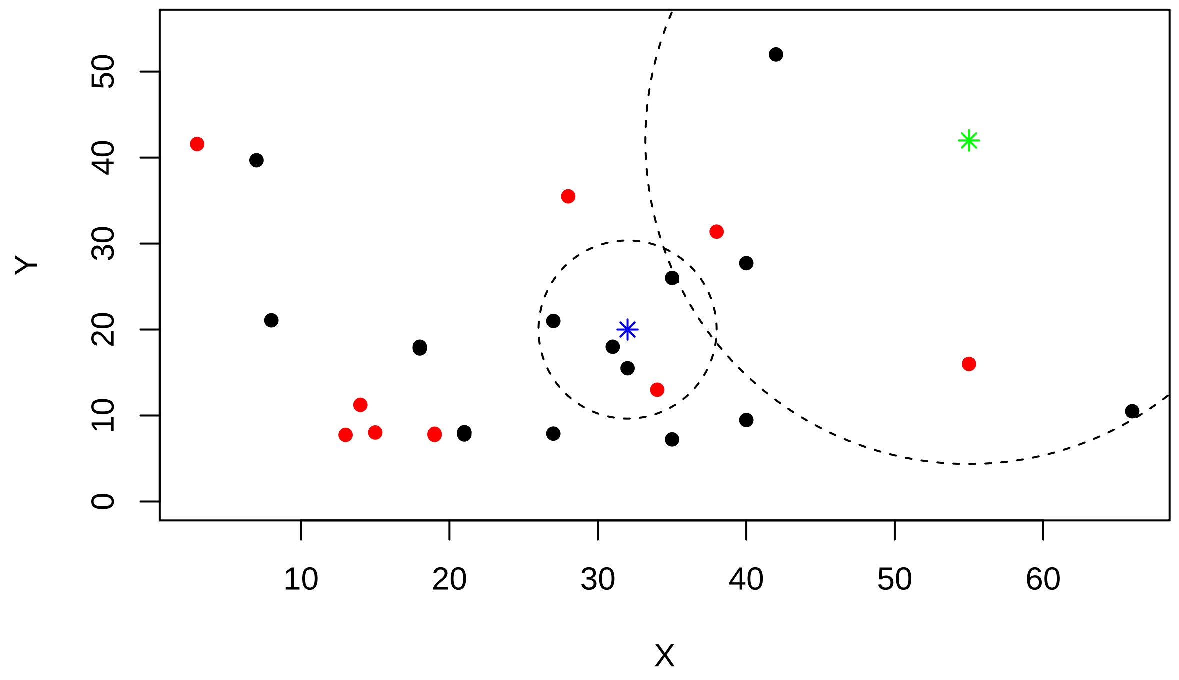
each of which has its advantages and drawbacks depending on the problem at hand. for instance in medecine if you are more interested to check the similarities among patients at some level then the first option would be your choice but you may not be able to predict new cases that are not similar to any of the training cases constrained by this threshold value. In contrast, if your goal is to predict all your new cases then you would opt for the second option, but with the cost of including neighbours that are far a way constrained by the fixed number of neighbours.

```
library(plotrix)
X<-c(55,31,35,34,15,28,8,38,35,19,27,40,39,19,66,28,42,21,18,14,40,27,3,19,21,32,13,18,7,21,49)
Y<-c(16,18,26,13,8.0292,35.5,21.075,31.3875,7.225,263,7.8958,27.7208,146.5208,7.75,10.5,82.1708,52,8.05
Z<-factor(c(1,0,0,1,1,1,0,1,0,0,0,0,1,1,0,0,0,0,0,1,0,0,1,1,0,0,1,0,0,0,1))
plot(X,Y,col=Z,ylim = c(0,55),pch=16)
points(x=32,y=20,col="blue",pch=8)
draw.circle(x=32,y=20,nv=1000,radius = 6,lty=2)
points(x=55,y=42,col="green",pch=8)
draw.circle(x=55,y=42,nv=1000,radius = 6,lty=2)
```



Using the fake data above to illustrate the difference between the two methods. Setting the threshold at 6 we see the blue dot has 5 neighbours and this dot would be predicted as black using the majority vote. However, with this threshold the green dot does not have any neighbour around and hence it will be left without prediction.

```
plot(X,Y,pch=16,col=Z,ylim = c(0,55))
points(x=32,y=20,col="blue",pch=8)
points(x=55,y=42,col="green",pch=8)
draw.circle(x=55,y=42,nv=1000,radius = 21.8,lty=2)
draw.circle(x=32,y=20,nv=1000,radius = 6,lty=2)
```



In contrast to the above plot the green dot can be predicted as black since it has 5 neighbours in whcih 3 are block, but this prediction casts doubt about its quality since all the neighbours are far a way from the dot of interest, and this is the major drawback of this method.

Note: In fact we can overcome the drawbacks of each method using the hyperparameters. To get predictions of all the new cases we can increase the distance threshold using the first method so that all the cases will be predicted (but we may lose accuracy if we have any single outlier). And we can reduce the threshold for the nearest neighbours number to get meaningful similarities (but we may lose accuracy with dots spread out from each other).

the similarity based on distance

To show the neighbours of each point we use the function **ml_approx_similarity_join** but this requires that the data has an **id** column, this is thus the reason why we have created this id before.

```
approx_join<-ml_approx_similarity_join(model_lsh,trained,
                                       trained,
                                       threshold=1,
                                     dist_col="dist")
approx_join
```

```
# Source: spark<?> [?? x 3]
    id_a  id_b  dist
   <int> <int> <dbl>
 1     6    10 0.236
 2     6     6 0
 3    13    23 0.552
 4    13     9 0.960
 5     13    13 0
 6     14    14 0
 7     25    25 0
 8     31    45 0.683
 9     49    50 0.468
10     58    61 0.819
# ... with more rows
```

This function joined the data **trained** with itself to get the similar observations. The threshold determine the value from which we consider two observations as similar. let's for instance pick up some similar observations and check out how they are similar.

```
train%>%
  filter(id %in% c(31,45,20))
```

```
# Source: spark<?> [?? x 9]
     id Survived Pclass Sex      Age SibSp Parch  Fare Embarked
  <int>    <int>  <int> <chr>  <dbl> <int> <int> <dbl> <chr>
1    31        1      1 female    30     0     0  93.5 S
2    20        1      1 female    26     0     0  78.8 S
3    45        1      1 female    38     0     0  80   S
```

As we see these passengers are all survived females in the same class without children or parents or siblings embarked from the same ports, their ages are more or less close to each other. Since they are embarked from the same port it is highly likely to be friends.

```
hashed<-ml_predict(model_lsh,tested)%>%
  ml_approx_similarity_join(model_lsh,
                            trained,
                            .,
                            threshold=1,
                            dist_col="dist")
hashed
```

```
# Source: spark<?> [?? x 3]
    id_a  id_b  dist
   <int> <int> <dbl>
 1    14    10 0.821
 2    85    88 0.222
 3    90    88 0.411
 4    90    26 0.597
 5    90    23 0.202
 6    90    90 0.766
 7    90    27 0.801
 8    93    27 0.649
 9    99    26 0.569
10    99    95 0.496
# ... with more rows
```

we can now shoose a particular person, say id_b=88, and then find his similar persons in the training set. By using the majority vote we decide if that person is survived or not.

```
m<-88
ids_train<-hashed%>%
  filter(id_b==m)%>%
  pull(id_a)
df1<-train%>%
  filter(id %in% ids_train)
```

```
df2<-test%>%
  filter(id==m)
df<-sdf_bind_rows(df1,df2)
df
```

```
# Source: spark<?> [?? x 9]
      id Survived Pclass Sex     Age SibSp Parch  Fare Embarked
   <int>    <int>  <int> <chr> <dbl> <int> <int> <dbl> <chr>
 1    85        1      1 male     28     0     0  30.5 S
 2    87        1      1 male     28     0     0  35.5 S
 3    90        1      1 male     34     0     0  26.6 S
 4    93        1      1 male     36     0     0  26.3 S
 5   272        0      1 male     28     0     0  25.9 S
 6   274        0      1 male     28     0     0  26.6 S
 7   281        0      1 male     28     0     0  47.1 S
 8   282        0      1 male     28     0     0  52   S
 9    81        1      1 male     28     0     0  26.6 S
10   273        0      1 male     28     0     0  26   S
# ... with more rows
```

```
df%>%
  filter(id!=88)%>%
  select(Survived)%>%
  collect()%>%
  table()
```

```
## .
##  0  1
## 13  9
```

Using the majority vote this person will be classified as not survived since the non survived persons number (13) is larger than survived persons number (9), and hence this person is correctly classified.

The similarity based on the number of nearest neighbours

Using the same above steps but here with the function

ml_approx_nearest_neighbors we can predict any point. for example let's take our previous passenger n 88 in the testing set.

```
id_input <- tested %>%
  filter(id==88)%>%
    pull(scaled) %>%
  unlist()
id_input
```

```
##  [1]  0.0000000000  1.7510638166 -0.5220305772  0.7892313849 -0.0009767992
##  [6] -0.4719918401 -0.5677786380 -0.0451313829  0.5297556282 -0.4190324991
```

These are the values of all the standardized vectors in the column **scaled**.

```
knn<-ml_approx_nearest_neighbors(
  model_lsh,
  trained,
  key = id_input,
  dist_col = 'dist',
  num_nearest_neighbors = 7
)
knn
```

```
# Source: spark<?> [?? x 5]
     id Survived scaled     hash        dist
  <int>    <int> <list>     <list>     <dbl>
1    84        1 <dbl [10]> <list [5]> 0.221
2    85        1 <dbl [10]> <list [5]> 0.222
3   277        0 <dbl [10]> <list [5]> 0.222
4    81        1 <dbl [10]> <list [5]> 0.231
5    82        1 <dbl [10]> <list [5]> 0.231
6   274        0 <dbl [10]> <list [5]> 0.231
7   273        0 <dbl [10]> <list [5]> 0.234
```

```
n<-sdf_nrow(knn)
pred<-knn%>%select(Survived)%>%
  summarise(p=sum(Survived)/n)
pred
```

```
# Source: spark<?> [?? x 1]
      p
  <dbl>
1 0.571
```

Since this probability is greater than 0.5 we predict this passenger as survived but actually they are not and this contradicts the above prediction from the first method.

To get the accuracy of the whole testing set, sparklyr unfortunately does not have a function for this. That is why we use the following for loop, which requires a lot of computing time since at the end of each iteration we collect the results into R .

```
mypred<-numeric(0)

for (i in 1:sdf_nrow(tested)) {
  id_input <- tested %>%
  filter(id==i)%>%
    pull(scaled) %>%
  unlist()
knn<-ml_approx_nearest_neighbors(
  model_lsh,
  trained,
  key = id_input,
  dist_col = 'dist',
  num_nearest_neighbors = 7
)
n<-sdf_nrow(knn)
pred<-knn%>%select(Survived)%>%
  summarise(p=sum(Survived)/n)%>%
  collect()
mypred<-rbind(mypred,pred)
}
mypred
```

```
# A tibble: 200 x 1
       p
   <dbl>
```

```
 1 0.714
 2 1
 3 1
 4 0.857
 5 1
 6 1
 7 1
 8 1
 9 1
10 1
# ... with 190 more rows
```

Now first we convert the probabilities into class labels, next we join this data frame with the testing data, and finally we use the function **confusionmatrix** from **caret** package.

```r
tested_R<-tested%>%
  select(Survived)%>%
  collect()
new<-cbind(mypred,tested_R)%>%
  mutate(predicted=ifelse(p>0.5,"1","0"))
  caret::confusionMatrix(as.factor(new$Survived),as.factor(new$predicted))
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 80 41
         1 49 30

               Accuracy : 0.55
                 95% CI : (0.4782, 0.6202)
    No Information Rate : 0.645
    P-Value [Acc > NIR] : 0.9977

                  Kappa : 0.0416

 Mcnemar's Test P-Value : 0.4606

            Sensitivity : 0.6202
            Specificity : 0.4225
         Pos Pred Value : 0.6612
         Neg Pred Value : 0.3797
             Prevalence : 0.6450
         Detection Rate : 0.4000
   Detection Prevalence : 0.6050
      Balanced Accuracy : 0.5213

       'Positive' Class : 0
```

The accuracy rate is smaller than the No information rate value 0.625 if we would have predicted all the cases as class label 0. this may due to the uncorrect number chosen for the nearest neighbours or the model is not suitable for this data.