In [1]:

```
import findspark
findspark.init()
findspark.find()

import pyspark
from pyspark import SparkContext, SparkConf, SQLContext
from pyspark.sql import SparkSession

from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master("local[*]") \
    .appName("GenericAppName") \
    .getOrCreate()
sc=spark.sparkContext
sqlContext = SQLContext(sc)
```

```
C:\Spark\Spark\spark-3.3.0-bin-hadoop3\spark-3.3.0-bin-hadoop3\python\pyspark\sql\context.py:112: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
```

Task I

In [2]:

```
# Ingest data 2015-2022
from pyspark import SparkFiles

players_15 = spark.read.csv('FIFA_DATA/players_15.csv',header=True, inferSchema = True)
players_16 = spark.read.csv('FIFA_DATA/players_16.csv',header=True, inferSchema = True)
players_17 = spark.read.csv('FIFA_DATA/players_17.csv',header=True, inferSchema = True)
players_18 = spark.read.csv('FIFA_DATA/players_18.csv',header=True, inferSchema = True)
players_19 = spark.read.csv('FIFA_DATA/players_19.csv',header=True, inferSchema = True)
players_20 = spark.read.csv('FIFA_DATA/players_20.csv',header=True, inferSchema = True)
players_21 = spark.read.csv('FIFA_DATA/players_21.csv',header=True, inferSchema = True)
players_22 = spark.read.csv('FIFA_DATA/players_22.csv',header=True, inferSchema = True)

```

In [3]:
```
print(players_15.columns[55])
```

```
movement_agility
```

In [4]:
```
#Add new column for the year
from pyspark.sql.functions import lit

players_15 = players_15.withColumn('Year', lit(2015))
players_16 = players_16.withColumn('Year', lit(2016))
players_17 = players_17.withColumn('Year', lit(2017))
players_18 = players_18.withColumn('Year', lit(2018))
players_19 = players_19.withColumn('Year', lit(2019))
players_20 = players_20.withColumn('Year', lit(2020))
players_21 = players_21.withColumn('Year', lit(2021))
players_22 = players_22.withColumn('Year', lit(2022))

```

In [5]:
```
#Ensure every record can be uniquely identified
#combine fifa id and year as index
from pyspark.sql import functions as sf

players_15 = players_15.withColumn('data_ID', sf.concat(sf.col('sofifa_id'),sf.lit('_'), sf.col('Year')))
players_16 = players_16.withColumn('data_ID', sf.concat(sf.col('sofifa_id'),sf.lit('_'), sf.col('Year')))
players_17 = players_17.withColumn('data_ID', sf.concat(sf.col('sofifa_id'),sf.lit('_'), sf.col('Year')))
players_18 = players_18.withColumn('data_ID', sf.concat(sf.col('sofifa_id'),sf.lit('_'), sf.col('Year')))
players_19 = players_19.withColumn('data_ID', sf.concat(sf.col('sofifa_id'),sf.lit('_'), sf.col('Year')))
players_20 = players_20.withColumn('data_ID', sf.concat(sf.col('sofifa_id'),sf.lit('_'), sf.col('Year')))
players_21 = players_21.withColumn('data_ID', sf.concat(sf.col('sofifa_id'),sf.lit('_'), sf.col('Year')))
players_22 = players_22.withColumn('data_ID', sf.concat(sf.col('sofifa_id'),sf.lit('_'), sf.col('Year')))
```

In [7]:
```
# union the data to one dataset
data = players_15.union(players_16)
data = data.union(players_17)
data = data.union(players_18)
data = data.union(players_19)
data = data.union(players_20)
data = data.union(players_21)
data = data.union(players_22)
```

In [8]:

```
#ingest the dataset into postgres

db_properties={}
#update your db username
db_properties['username']="postgres"
#update your db password
db_properties['password']="20000823"
#make sure you got the right port number here
db_properties['url']= "jdbc:postgresql://localhost:5432/postgres"
#make sure you had the Postgres JAR file in the right location
db_properties['driver']="org.postgresql.Driver"
db_properties['table']= "fifa.data"



data.write.format("jdbc")\
.mode("overwrite")\
.option("url", db_properties['url'])\
.option("dbtable", db_properties['table'])\
.option("user", db_properties['username'])\
.option("password", db_properties['password'])\
.option("Driver", db_properties['driver'])\
.save()

```

In [9]:
```
1 data.printSchema()
```

```
|-- lwb: string (nullable = true)
|-- ldm: string (nullable = true)
|-- cdm: string (nullable = true)
|-- rdm: string (nullable = true)
|-- rwb: string (nullable = true)
|-- lb: string (nullable = true)
|-- lcb: string (nullable = true)
|-- cb: string (nullable = true)
|-- rcb: string (nullable = true)
|-- rb: string (nullable = true)
|-- gk: string (nullable = true)
|-- player_face_url: string (nullable = true)
|-- club_logo_url: string (nullable = true)
|-- club_flag_url: string (nullable = true)
|-- nation_logo_url: string (nullable = true)
|-- nation_flag_url: string (nullable = true)
|-- Year: integer (nullable = false)
|-- data_ID: string (nullable = true)
```

In [10]:

```
# read the data back from the postgres
data_read = sqlContext.read.format("jdbc")\
    .option("url", db_properties['url'])\
    .option("dbtable", db_properties['table'])\
    .option("user", db_properties['username'])\
    .option("password", db_properties['password'])\
    .option("Driver", db_properties['driver'])\
    .load()

data_read.show(1, vertical=True)
```

```
-RECORD 0-----------------------------
 lwb                  | 26+1
 ldm                  | 26+1
 cdm                  | 26+1
 rdm                  | 26+1
 rwb                  | 26+1
 lb                   | 26+1
 lcb                  | 27+1
 cb                   | 27+1
 rcb                  | 27+1
 rb                   | 26+1
 gk                   | 68+1
 player_face_url      | https://cdn.sofif... (https://cdn.sofif...)
 club_logo_url        | https://cdn.sofif... (https://cdn.sofif...)
 club_flag_url        | https://cdn.sofif... (https://cdn.sofif...)
 nation_logo_url      | null
 nation_flag_url      | https://cdn.sofif... (https://cdn.sofif...)
 Year                 | 2017
 data_ID              | 190745_2017
only showing top 1 row
```

Task II

In [11]:

```
#In order to find the club with highest numbers of players
#use where to find the data in 2022
#group the data by club name and sort them by the count of data
highest_number_club = data_read.where(data_read['Year']=='2022').groupBy("club_name").count().sort("count",ascending=False).li
print("club with highest numbers:",highest_number_club['club_name'].values)
```

```
club with highest numbers: ['Norwich City' 'FC Barcelona' 'ESTAC Troyes' 'RC Celta de Vigo'
 'RCD Espanyol de Barcelona' 'Newcastle United' 'Paris Saint-Germain'
 'Crystal Palace' 'Burnley' 'Granada CF' 'RCD Mallorca' 'Genoa'
 'Real Madrid CF' 'Manchester United' 'Arsenal' 'Brentford'
 'VfB Stuttgart' 'Real Betis Balompié' 'CA Osasuna' 'Villarreal CF'
 'Southampton' 'Leicester City' 'Venezia FC' 'Valencia CF'
 'Borussia Mönchengladbach' 'Tottenham Hotspur' 'Sevilla FC'
 'TSG Hoffenheim' 'Wolverhampton Wanderers' 'Brighton & Hove Albion'
 'Liverpool' 'Olympique de Marseille' 'Chelsea' 'Everton'
 'West Ham United']
```

In [12]:

```
# this kernel shows the detailed number of players in each club and we can have the highest number from this.
# we drew the barplot of the count to give a direct interpretation
import seaborn as sns
from IPython import display
import matplotlib.pyplot as plt

top_50_highest_number_club = data_read.where(data_read['Year']=='2022').groupBy("club_name").count().sort("count",ascending=Fa
print(top_50_highest_number_club)
plt.figure( figsize = ( 15, 8 ) )
sns.barplot( x="club_name", y="count", data=highest_number_club)
plt.show()
```

	club_name	count
0	None	61
1	RC Celta de Vigo	33
2	RCD Espanyol de Barcelona	33
3	FC Barcelona	33
4	RCD Mallorca	33
5	Genoa	33
6	Real Madrid CF	33
7	ESTAC Troyes	33
8	Norwich City	33
9	VfB Stuttgart	33
10	Real Betis Balompié	33
11	Newcastle United	33
12	Arsenal	33
13	Brentford	33
14	Manchester United	33
15	Tottenham Hotspur	33
16	Sevilla FC	33
17	TSG Hoffenheim	33

In [14]:

```
# the number of players older than 27 years oldfor each club
# we used where to find the players who are older than 27 and group the data by club_name, then we sort the data
# to find the club who has the highest number
# we sum the age for each club

import numpy as np
highest_number_over27_count = data_read.where(data_read['age']>27).groupBy("club_name").count().sort("count",ascending=False).
highest_number_over27_sum = data_read.where(data_read['age']>27).groupBy("club_name").sum('age').toPandas()
# highest_number_over27_value = highest_number_over27['age'].values
# .sort("count",ascending=False).limit(50).toPandas()
print(highest_number_over27_count)
print(highest_number_over27_sum)
average_over27 = highest_number_over27_sum['sum(age)'].values/highest_number_over27_count['count'].values
print(type(average_over27),average_over27)
```

```
                  club_name  count
0                      None    874
1     İstanbul Başakşehir FK    133
2     Jeonbuk Hyundai Motors    118
3        FC Lokomotiv Moscow    108
4             Crystal Palace    106
...                      ...    ...
1007     Caracas Fútbol Club      3
1008          SC Freiburg II      2
1009            FC Helsingør      2
1010            FC Dordrecht      1
1011    Borussia Dortmund II      1

[1012 rows x 2 columns]
                      club_name  sum(age)
0                       Palermo      1105
1     CD Everton de Viña del Mar      1014
2                 Shonan Bellmare      1676
3                     Yeovil Town       847
```

In [15]:

```
# we calculated the average and found the highest average
highest_number_over27_avg = data_read.where(data_read['age']>27).groupBy("club_name").avg('age').sort("avg(age)",ascending=Fal
print(highest_number_over27_avg)
```

	club_name	avg(age)
0	Yokohama FC	34.703704
1	Wexford Youths	34.000000
2	Zamora Fútbol Club	33.857143
3	Centro Atlético Fénix	33.600000
4	CF Fuenlabrada	33.545455

In [16]:

```
top1_highest_number_over27_avg = data_read.where(data_read['age']>27).groupBy("club_name").avg('age').sort("avg(age)",ascendin
print("club with highest numbers over 27:",top1_highest_number_over27_avg['club_name'].values)
```

```
club with highest numbers over 27: ['Yokohama FC']
```

In [17]:

```
# most frequent nation_positions for each year
# we also used where to filter the year, and group them by nation_position, count the number of each position and sort
# them to search the highest one.
most_frequent_nation_2015 = data_read.where(data_read['Year']==2015).groupBy("nation_position").count().sort("count",ascending
print("Most frequent nation position 2015:",most_frequent_nation_2015['nation_position'].values)
most_frequent_nation_2016 = data_read.where(data_read['Year']==2016).groupBy("nation_position").count().sort("count",ascending
print("Most frequent nation position 2016:",most_frequent_nation_2016['nation_position'].values)
most_frequent_nation_2017 = data_read.where(data_read['Year']==2017).groupBy("nation_position").count().sort("count",ascending
print("Most frequent nation position 2017:",most_frequent_nation_2017['nation_position'].values)
most_frequent_nation_2018 = data_read.where(data_read['Year']==2018).groupBy("nation_position").count().sort("count",ascending
print("Most frequent nation position 2018:",most_frequent_nation_2018['nation_position'].values)
most_frequent_nation_2019 = data_read.where(data_read['Year']==2019).groupBy("nation_position").count().sort("count",ascending
print("Most frequent nation position 2019:",most_frequent_nation_2019['nation_position'].values)
most_frequent_nation_2020 = data_read.where(data_read['Year']==2020).groupBy("nation_position").count().sort("count",ascending
print("Most frequent nation position 2020:",most_frequent_nation_2020['nation_position'].values)
most_frequent_nation_2021 = data_read.where(data_read['Year']==2021).groupBy("nation_position").count().sort("count",ascending
print("Most frequent nation position 2021:",most_frequent_nation_2021['nation_position'].values)
most_frequent_nation_2022 = data_read.where(data_read['Year']==2022).groupBy("nation_position").count().sort("count",ascending
print("Most frequent nation position 2022:",most_frequent_nation_2022['nation_position'].values)
```

```
Most frequent nation position 2015: ['SUB']
Most frequent nation position 2016: ['SUB']
Most frequent nation position 2017: ['SUB']
Most frequent nation position 2018: ['SUB']
Most frequent nation position 2019: ['SUB']
Most frequent nation position 2020: ['SUB']
Most frequent nation position 2021: ['SUB']
Most frequent nation position 2022: ['SUB']
```

In [83]:

```
# most_frequent_nation_2015 = data_read.where(data_read['Year']==2015).groupBy("nation_position").count().sort("count",ascendi
# print(most_frequent_nation_2015)
```

Task III

Data Engineering and cleaning

In [19]:
```
# Since we only use the skillsets of the players, we dropped all the unneccessary columns
drop_list = ['player_position',' nationality_name','work_rate','body_type','real_face','data_ID','short_name','long_name','dob
                            'club_loaned_from','club_joined','club_contract_valid_until',
                            'nationality_id','nation_team_id','nation_position','nation_jersey_number',
                            'release_clause_eur','player_tags','player_traits','player_face_url',
                            'club_logo_url','club_flag_url','nation_logo_url','nation_flag_url',
                            'player_url','mentality_composure','goalkeeping_speed','club_position','league_level','league_name
```

In [20]:
```
print(len(drop_list))
```

```
33
```

In [21]:
```
df_read = data_read.drop('player_positions','nationality_name','work_rate','body_type','real_face','data_ID','short_name','lon
                            'club_loaned_from','club_joined','club_contract_valid_until',
                            'nationality_id','nation_team_id','nation_position','nation_jersey_number',
                            'release_clause_eur','player_tags','player_traits','player_face_url',
                            'club_logo_url','club_flag_url','nation_logo_url','nation_flag_url',
                            'player_url','mentality_composure','goalkeeping_speed','club_position','league_level','league_name
```

In [22]:
```
print(len(data_read.columns))
print(len(df_read.columns))
```

```
112
79
```

In [23]:
```
df_read.printSchema()
```

```
root
 |-- sofifa_id: integer (nullable = true)
 |-- overall: integer (nullable = true)
 |-- potential: integer (nullable = true)
 |-- value_eur: double (nullable = true)
 |-- wage_eur: double (nullable = true)
 |-- age: integer (nullable = true)
 |-- height_cm: integer (nullable = true)
 |-- weight_kg: integer (nullable = true)
 |-- preferred_foot: string (nullable = true)
 |-- weak_foot: integer (nullable = true)
 |-- skill_moves: integer (nullable = true)
 |-- international_reputation: integer (nullable = true)
 |-- pace: integer (nullable = true)
 |-- shooting: integer (nullable = true)
 |-- passing: integer (nullable = true)
 |-- dribbling: integer (nullable = true)
 |-- defending: integer (nullable = true)
 |-- physic: integer (nullable = true)
```

In [85]:
```

# from pyspark.sql.functions import *

# # Notice I dropped isPenalty and isSTPlay
# null_counts_plays_df = df_read.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) \
#                         for c in df_read.columns])

# null_counts_plays_df.show(truncate=False, vertical=True)
```

impute 'pace','shooting','passing','dribbling','defending','physic'

In [ ]:
```
# we checked the missing values of the whole dataset and we found out that the columns with
# larger amoungt of missing value: 'pace','shooting','passing','dribbling','defending','physic'
# Because there were many missing value in the data, we want to impute the missing value with the median value
```

In [24]:

```
from pyspark.ml.feature import Imputer
columns_to_be_imputed = ['pace']
value_not_in_dataset = -200

# Replace None/Missing Value with a value that can't be present in the dataset.
df_with_filled_na = df_read.fillna(-200, columns_to_be_imputed)

#Create new columns with imputed values. New columns will be suffixed with "_imputed"
imputer = Imputer (
            inputCols=columns_to_be_imputed,
            outputCols=["{}_imputed".format(c) for c in columns_to_be_imputed])\
            .setStrategy("median").setMissingValue(value_not_in_dataset)

df_imputed = imputer.fit(df_with_filled_na).transform(df_with_filled_na)
# we will drop the old column without imputation. We have only one column to be imputed
df_imputed_enhanced = df_imputed.drop(columns_to_be_imputed[0])

# We will rename our newly imputed column with the correct name
df_fully_imputed = df_imputed_enhanced.withColumnRenamed("pace_imputed","pace")
```

In [25]:

```
from pyspark.ml.feature import Imputer
columns_to_be_imputed = ['shooting']
value_not_in_dataset = -200

# Replace None/Missing Value with a value that can't be present in the dataset.
df_with_filled_na2 = df_fully_imputed.fillna(-200, columns_to_be_imputed)

#Create new columns with imputed values. New columns will be suffixed with "_imputed"
imputer = Imputer (
            inputCols=columns_to_be_imputed,
            outputCols=["{}_imputed".format(c) for c in columns_to_be_imputed])\
            .setStrategy("median").setMissingValue(value_not_in_dataset)

df_imputed2 = imputer.fit(df_with_filled_na2).transform(df_with_filled_na2)
# we will drop the old column without imputation. We have only one column to be imputed
df_imputed_enhanced2 = df_imputed2.drop(columns_to_be_imputed[0])

# We will rename our newly imputed column with the correct name
df_fully_imputed2 = df_imputed_enhanced2.withColumnRenamed("shooting_imputed","shooting")
```

In [26]:

```
from pyspark.ml.feature import Imputer
columns_to_be_imputed = ['passing']
value_not_in_dataset = -200

# Replace None/Missing Value with a value that can't be present in the dataset.
df_with_filled_na3 = df_fully_imputed2.fillna(-200, columns_to_be_imputed)

#Create new columns with imputed values. New columns will be suffixed with "_imputed"
imputer = Imputer (
            inputCols=columns_to_be_imputed,
            outputCols=["{}_imputed".format(c) for c in columns_to_be_imputed])\
            .setStrategy("median").setMissingValue(value_not_in_dataset)

df_imputed3 = imputer.fit(df_with_filled_na3).transform(df_with_filled_na3)
# we will drop the old column without imputation. We have only one column to be imputed
df_imputed_enhanced3 = df_imputed3.drop(columns_to_be_imputed[0])

# We will rename our newly imputed column with the correct name
df_fully_imputed3 = df_imputed_enhanced3.withColumnRenamed("passing_imputed","passing")
```

In [27]:

```
from pyspark.ml.feature import Imputer
columns_to_be_imputed = ['dribbling']
value_not_in_dataset = -200

# Replace None/Missing Value with a value that can't be present in the dataset.
df_with_filled_na4 = df_fully_imputed3.fillna(-200, columns_to_be_imputed)

#Create new columns with imputed values. New columns will be suffixed with "_imputed"
imputer = Imputer (
            inputCols=columns_to_be_imputed,
            outputCols=["{}_imputed".format(c) for c in columns_to_be_imputed])\
            .setStrategy("median").setMissingValue(value_not_in_dataset)

df_imputed4 = imputer.fit(df_with_filled_na4).transform(df_with_filled_na4)
# we will drop the old column without imputation. We have only one column to be imputed
df_imputed_enhanced4 = df_imputed4.drop(columns_to_be_imputed[0])

# We will rename our newly imputed column with the correct name
df_fully_imputed4 = df_imputed_enhanced4.withColumnRenamed("dribbling_imputed","dribbling")
```

In [28]:

```
from pyspark.ml.feature import Imputer
columns_to_be_imputed = ['defending']
value_not_in_dataset = -200

# Replace None/Missing Value with a value that can't be present in the dataset.
df_with_filled_na5 = df_fully_imputed4.fillna(-200, columns_to_be_imputed)

#Create new columns with imputed values. New columns will be suffixed with "_imputed"
imputer = Imputer (
            inputCols=columns_to_be_imputed,
            outputCols=["{}_imputed".format(c) for c in columns_to_be_imputed])\
            .setStrategy("median").setMissingValue(value_not_in_dataset)

df_imputed5 = imputer.fit(df_with_filled_na5).transform(df_with_filled_na5)
# we will drop the old column without imputation. We have only one column to be imputed
df_imputed_enhanced5 = df_imputed5.drop(columns_to_be_imputed[0])

# We will rename our newly imputed column with the correct name
df_fully_imputed5 = df_imputed_enhanced5.withColumnRenamed("defending_imputed","defending")
```

In [29]:

```
from pyspark.ml.feature import Imputer
columns_to_be_imputed = ['physic']
value_not_in_dataset = -200

# Replace None/Missing Value with a value that can't be present in the dataset.
df_with_filled_na6 = df_fully_imputed5.fillna(-200, columns_to_be_imputed)

#Create new columns with imputed values. New columns will be suffixed with "_imputed"
imputer = Imputer (
            inputCols=columns_to_be_imputed,
            outputCols=["{}_imputed".format(c) for c in columns_to_be_imputed])\
            .setStrategy("median").setMissingValue(value_not_in_dataset)

df_imputed6 = imputer.fit(df_with_filled_na6).transform(df_with_filled_na6)
# we will drop the old column without imputation. We have only one column to be imputed
df_imputed_enhanced6 = df_imputed6.drop(columns_to_be_imputed[0])

# We will rename our newly imputed column with the correct name
df_fully_imputed6 = df_imputed_enhanced6.withColumnRenamed("physic_imputed","physic")
```

In [30]:
```
from pyspark.sql.functions import *

# # Notice I dropped isPenalty and isSTPlay
# null_counts_plays_df = df_fully_imputed6.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) \
#                         for c in df_read.columns])

# null_counts_plays_df.show(truncate=False, vertical=True)
```

drop null

In [ ]:
```
# we chaecked the missing value again and found out the rows with missing value
# Then we drop the rows
# Reason: The number of row was small compared to the entire dataset, so we just drop them
```

In [31]:
```
df_read = df_fully_imputed6.dropna()
```

In [32]:
```
null_counts_plays_df = df_read.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) \
                        for c in df_read.columns])

null_counts_plays_df.show(truncate=False, vertical=True)
```

```
-RECORD 0-------------------------
 sofifa_id                   | 0
 overall                     | 0
 potential                   | 0
 value_eur                   | 0
 wage_eur                    | 0
 age                         | 0
 height_cm                   | 0
 weight_kg                   | 0
 preferred_foot              | 0
 weak_foot                   | 0
 skill_moves                 | 0
 international_reputation    | 0
 attacking_crossing          | 0
 attacking_finishing         | 0
 attacking_heading_accuracy  | 0
 attacking_short_passing     | 0
 attacking_volleys           | 0
 skill_dribbling             | 0
```

Data preprocessing

label to binary

In [33]:
```
# Because there are only two values in the 'preferred_foot' column, we convert the value to 0 and 1
label_to_binary = udf(lambda name: 0.0 if name == 'Left' else 1.0)
df_read_bi = df_read.withColumn('preferred_foot', label_to_binary(col('preferred_foot')))
```

In [34]:
```
# df_read_bi.show(vertical = True)
```

string to int

In [35]:
```
# We found the format "XX+X" in some columns and we calculated them and converted them to int
def addall(string):
    if len(string)== 4:
        string = int(string[0:2])+int(string[-1])
    elif len(string) ==2:
        string = int(string)
    elif len(string)== 6:
        string = int(string[-2:])
    return string
```

In [36]:
```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType
```

In [37]:
```
addallUDF = udf(lambda i:addall(i), IntegerType())
```

In [38]:
```
str_to_int_list = ['ls','st','rs','lw','lf','cf','rf','rw','lam','cam','ram','lm','lcm','cm','rcm','rm','lwb',
                   'ldm','cdm','rdm','rwb','lb','lcb','cb','rcb','rb','gk']
```

In [39]:

```
df_read_int = df_read_bi.withColumn('ls', addallUDF(col('ls')))
df_read_int = df_read_int.withColumn('st', addallUDF(col('st')))
df_read_int = df_read_int.withColumn('rs', addallUDF(col('rs')))
df_read_int = df_read_int.withColumn('lw', addallUDF(col('lw')))
df_read_int = df_read_int.withColumn('lf', addallUDF(col('lf')))
df_read_int = df_read_int.withColumn('cf', addallUDF(col('cf')))
df_read_int = df_read_int.withColumn('rf', addallUDF(col('rf')))
df_read_int = df_read_int.withColumn('rw', addallUDF(col('rw')))
df_read_int = df_read_int.withColumn('lam', addallUDF(col('lam')))
df_read_int = df_read_int.withColumn('cam', addallUDF(col('cam')))
df_read_int = df_read_int.withColumn('ram', addallUDF(col('ram')))
df_read_int = df_read_int.withColumn('lm', addallUDF(col('lm')))
df_read_int = df_read_int.withColumn('lcm', addallUDF(col('lcm')))
df_read_int = df_read_int.withColumn('cm', addallUDF(col('cm')))
df_read_int = df_read_int.withColumn('rcm', addallUDF(col('rcm')))
df_read_int = df_read_int.withColumn('rm', addallUDF(col('rm')))
df_read_int = df_read_int.withColumn('lwb', addallUDF(col('lwb')))
df_read_int = df_read_int.withColumn('ldm', addallUDF(col('ldm')))
df_read_int = df_read_int.withColumn('cdm', addallUDF(col('cdm')))
df_read_int = df_read_int.withColumn('rdm', addallUDF(col('rdm')))
df_read_int = df_read_int.withColumn('rwb', addallUDF(col('rwb')))
df_read_int = df_read_int.withColumn('lb', addallUDF(col('lb')))
df_read_int = df_read_int.withColumn('lcb', addallUDF(col('lcb')))
df_read_int = df_read_int.withColumn('cb', addallUDF(col('cb')))
df_read_int = df_read_int.withColumn('rcb', addallUDF(col('rcb')))
df_read_int = df_read_int.withColumn('rb', addallUDF(col('rb')))
df_read_int = df_read_int.withColumn('gk', addallUDF(col('gk')))
```

cast data type

In [40]:
```
print(df_read_int.columns)
```

```
['sofifa_id', 'overall', 'potential', 'value_eur', 'wage_eur', 'age', 'height_cm', 'weight_kg', 'preferred_foot', 'weak_foot', 's
kill_moves', 'international_reputation', 'attacking_crossing', 'attacking_finishing', 'attacking_heading_accuracy', 'attacking_sh
ort_passing', 'attacking_volleys', 'skill_dribbling', 'skill_curve', 'skill_fk_accuracy', 'skill_long_passing', 'skill_ball_contr
ol', 'movement_acceleration', 'movement_sprint_speed', 'movement_agility', 'movement_reactions', 'movement_balance', 'power_shot_
power', 'power_jumping', 'power_stamina', 'power_strength', 'power_long_shots', 'mentality_aggression', 'mentality_interception
s', 'mentality_positioning', 'mentality_vision', 'mentality_penalties', 'defending_marking_awareness', 'defending_standing_tackl
e', 'defending_sliding_tackle', 'goalkeeping_diving', 'goalkeeping_handling', 'goalkeeping_kicking', 'goalkeeping_positioning',
'goalkeeping_reflexes', 'ls', 'st', 'rs', 'lw', 'lf', 'cf', 'rf', 'rw', 'lam', 'cam', 'ram', 'lm', 'lcm', 'cm', 'rcm', 'rm', 'lw
b', 'ldm', 'cdm', 'rdm', 'rwb', 'lb', 'lcb', 'cb', 'rcb', 'rb', 'gk', 'Year', 'pace', 'shooting', 'passing', 'dribbling', 'defend
ing', 'physic']
```

In [ ]:
```
# we changed the type of data to DoubleType
```

In [41]:
```
from pyspark.sql.types import DoubleType
for i in df_read_int.columns:
    df_read_int = df_read_int.withColumn(i, df_read_int[i].cast(DoubleType()))
```

In [42]:
```
df_read_int.printSchema()
```

```
root
 |-- sofifa_id: double (nullable = true)
 |-- overall: double (nullable = true)
 |-- potential: double (nullable = true)
 |-- value_eur: double (nullable = true)
 |-- wage_eur: double (nullable = true)
 |-- age: double (nullable = true)
 |-- height_cm: double (nullable = true)
 |-- weight_kg: double (nullable = true)
 |-- preferred_foot: double (nullable = true)
 |-- weak_foot: double (nullable = true)
 |-- skill_moves: double (nullable = true)
 |-- international_reputation: double (nullable = true)
 |-- attacking_crossing: double (nullable = true)
 |-- attacking_finishing: double (nullable = true)
 |-- attacking_heading_accuracy: double (nullable = true)
 |-- attacking_short_passing: double (nullable = true)
 |-- attacking_volleys: double (nullable = true)
 |-- skill_dribbling: double (nullable = true)
```

In [43]:
```
df_read_features = df_read_int.withColumn('overall_new',col('overall')).drop('overall')
```

In [44]:
```
df_read_int.show(5,vertical = True)
```

```
-RECORD 0--------------------------------
 sofifa_id                   | 190745.0
 overall                     | 69.0
 potential                   | 74.0
 value_eur                   | 1100000.0
 wage_eur                    | 10000.0
 age                         | 25.0
 height_cm                   | 191.0
 weight_kg                   | 80.0
 preferred_foot              | 1.0
 weak_foot                   | 2.0
 skill_moves                 | 1.0
 international_reputation    | 1.0
 attacking_crossing          | 12.0
 attacking_finishing         | 20.0
 attacking_heading_accuracy  | 12.0
 attacking_short_passing     | 22.0
 attacking_volleys           | 20.0
 skill_dribbling             | 18.0
```

In [45]:
```
feature_cols = df_read_int.columns
del feature_cols[2]
```

In [46]:
```
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors
```

In [47]:
```
len(df_read_features.columns)
```

Out[47]: 79

In [48]:
```
null_counts_plays_df_2 = df_read_features.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) \
                         for c in df_read_features.columns])

null_counts_plays_df_2.show(truncate=False, vertical=True)
```

```
lwb                  | 0
ldm                  | 0
cdm                  | 0
rdm                  | 0
rwb                  | 0
lb                   | 0
lcb                  | 1
cb                   | 1
rcb                  | 1
rb                   | 0
gk                   | 451
Year                 | 0
pace                 | 0
shooting             | 0
passing              | 0
dribbling            | 0
defending            | 0
physic               | 0
overall_new          | 0
```

In [49]:
```
df_read_features = df_read_features.dropna()
```

In [50]:

```
# we checked the data again to ensure there is no missing value in the dataset
null_counts_plays_df_2 = df_read_features.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) \
                         for c in df_read_features.columns])

null_counts_plays_df_2.show(truncate=False, vertical=True)
```

```
-RECORD 0--------------------------
 sofifa_id                   | 0
 potential                   | 0
 value_eur                   | 0
 wage_eur                    | 0
 age                         | 0
 height_cm                   | 0
 weight_kg                   | 0
 preferred_foot              | 0
 weak_foot                   | 0
 skill_moves                 | 0
 international_reputation    | 0
 attacking_crossing          | 0
 attacking_finishing         | 0
 attacking_heading_accuracy  | 0
 attacking_short_passing     | 0
 attacking_volleys           | 0
 skill_dribbling             | 0
 skill_curve                 | 0
```

In [ ]:

```
# we assembled the features in the data to a vector so that it can be recognized by pyspark models
```

In [51]:
```
def transData(data):
    return data.rdd.map(lambda r: [r[-1], Vectors.dense(r[:-1])]).\
            toDF(['output','features'])

data= transData(df_read_features)
data.show()
```

```
+------+--------------------+
|output|            features|
+------+--------------------+
|  69.0|[190745.0,74.0,11...|
|  69.0|[190780.0,73.0,10...|
|  69.0|[190783.0,69.0,65...|
|  69.0|[190792.0,72.0,10...|
|  69.0|[190883.0,69.0,77...|
|  69.0|[191079.0,73.0,11...|
|  69.0|[191096.0,70.0,95...|
|  69.0|[191135.0,72.0,12...|
|  69.0|[191222.0,71.0,10...|
|  69.0|[191252.0,69.0,10...|
|  69.0|[191443.0,69.0,32...|
|  69.0|[191548.0,69.0,65...|
|  69.0|[191679.0,69.0,72...|
|  69.0|[191834.0,69.0,82...|
|  69.0|[192009.0,76.0,12...|
|  69.0|[192179.0,69.0,87...|
|  69.0|[192297.0,69.0,87...|
|  69.0|[192449.0,72.0,12...|
|  69.0|[192457.0,70.0,10...|
|  69.0|[192473.0,69.0,11...|
+------+--------------------+
only showing top 20 rows
```

In [ ]:
```
# we normalized the features by using StandardScaler
```

In [52]:
```
from pyspark.ml.feature import StandardScaler
Scalerizer=StandardScaler().setInputCol("features").setOutputCol("norm_features")
data_norm = Scalerizer.fit(data).transform(data)
```

In [53]:
```
data_norm = data_norm.drop('features')
data_norm = data_norm.withColumn('features',col('norm_features')).drop('norm_features')
```

In [54]:
```
# data_norm.select('features').show(vertical = False)
```

In [55]:
```
# (trainingData, testData) = data_norm.randomSplit([0.8, 0.2])

# then we splited the data in to training and test dataset

(trainingData, testData) = data_norm.randomSplit([0.8, 0.2])
```

In [56]:
```
trainingData.describe().show()
```

```
+-------+-----------------+
|summary|           output|
+-------+-----------------+
|  count|           111935|
|   mean|65.68846205387055|
| stddev|7.070859372775934|
|    min|             40.0|
|    max|             94.0|
+-------+-----------------+
```

In [57]:
```
print("Training Dataset Count: " + str(trainingData.count()))
print("Test Dataset Count: " + str(testData.count()))
```

```
Training Dataset Count: 111935
Test Dataset Count: 27795
```

Pyspark

Random Forest

In [58]:
```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

rf2 = RandomForestRegressor(featuresCol = 'features', labelCol = 'output',numTrees=200, maxDepth=5)

# Train model.  This also runs the indexer.
model2 = rf2.fit(trainingData)

# Make predictions.
predictions2 = model2.transform(testData)

predictions2.show(5)
```

```
+------+--------------------+-----------------+
|output|            features|       prediction|
+------+--------------------+-----------------+
|  41.0|[6.53690364271165...|53.49295481506021|
|  44.0|[6.41599749220878...|53.49295481506021|
|  44.0|[6.58719665904405...|53.49295481506021|
|  44.0|[6.60650222035666...|53.80888048666465|
|  45.0|[6.17380835592218...|53.55483362781141|
+------+--------------------+-----------------+
only showing top 5 rows
```

In [59]:
```
evaluator2 = RegressionEvaluator(
    labelCol="output", predictionCol="prediction", metricName="rmse")
rmse2 = evaluator2.evaluate(predictions2)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse2)
```

```
Root Mean Squared Error (RMSE) on test data = 1.58138
```

tune the parameters

In [60]:

```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

rf3 = RandomForestRegressor(featuresCol = 'features', labelCol = 'output',numTrees=100, maxDepth=5)

# Train model.  This also runs the indexer.
model3 = rf3.fit(trainingData)

# Make predictions.
predictions3 = model3.transform(testData)

predictions3.show(5)

evaluator3 = RegressionEvaluator(
    labelCol="output", predictionCol="prediction", metricName="rmse")
rmse3 = evaluator3.evaluate(predictions3)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse3)
```

```
+------+--------------------+------------------+
|output|            features|        prediction|
+------+--------------------+------------------+
|  41.0|[6.53690364271165...|53.512072880945354|
|  44.0|[6.41599749220878...|53.512072880945354|
|  44.0|[6.58719665904405...|53.512072880945354|
|  44.0|[6.60650222035666...| 53.76495696951452|
|  45.0|[6.17380835592218...| 53.60100459946284|
+------+--------------------+------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 1.60538
```

In [61]:
```
1 trainingData.show(5)
```

```
+------+--------------------+
|output|            features|
+------+--------------------+
|  40.0|[6.30738196932836...|
|  40.0|[6.46263810504961...|
|  41.0|[6.40057623302213...|
|  42.0|[6.12293559300381...|
|  42.0|[6.22430428355967...|
+------+--------------------+
only showing top 5 rows
```

In [62]:

```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

rf_1 = RandomForestRegressor(featuresCol = 'features', labelCol = 'output',numTrees=100, maxDepth=6)

# Train model.  This also runs the indexer.
model_1 = rf_1.fit(trainingData)

# Make predictions.
predictions_1 = model_1.transform(testData)

predictions_1.show(5)

evaluator_1 = RegressionEvaluator(
    labelCol="output", predictionCol="prediction", metricName="rmse")
rmse_1 = evaluator_1.evaluate(predictions_1)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_1)
```

```
+------+--------------------+-----------------+
|output|            features|       prediction|
+------+--------------------+-----------------+
|  41.0|[6.53690364271165...|52.39016251553484|
|  44.0|[6.41599749220878...|52.39016251553484|
|  44.0|[6.58719665904405...|52.39016251553484|
|  44.0|[6.60650222035666...|52.77358279353466|
|  45.0|[6.17380835592218...|52.47513371165757|
+------+--------------------+-----------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 1.33804
```

In [ ]:

```
1
```

In [63]:
```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.evaluation import BinaryClassificationEvaluator

from sklearn.metrics import roc_curve
import pyspark.sql.functions as F
import pyspark.sql.types as T
import numpy
from matplotlib import pyplot as plt

from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.sql.types import Row
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import MultilayerPerceptronClassifier
```

Linear Regression

In [ ]:
```

```

In [64]:
```
from pyspark.ml.regression import LinearRegression

lrg = LinearRegression(featuresCol = 'features', labelCol = 'output')
# maxIter: int = 100 regParam=0.0
model5 = lrg.fit(trainingData)
```

In [65]:

```
predictions5 = model5.transform(testData)

predictions5.show(5)

evaluator5 = RegressionEvaluator(
    labelCol="output", predictionCol="prediction", metricName="rmse")
rmse5 = evaluator5.evaluate(predictions5)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse5)
```

```
+------+--------------------+------------------+
|output|            features|        prediction|
+------+--------------------+------------------+
|  41.0|[6.53690364271165...|40.753436154756855|
|  44.0|[6.41599749220878...| 45.19423746263686|
|  44.0|[6.58719665904405...|48.668658566473596|
|  44.0|[6.60650222035666...|44.114411895386354|
|  45.0|[6.17380835592218...| 45.62358471954002|
+------+--------------------+------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 1.82633
```

tune

In [66]:

```
from pyspark.ml.regression import LinearRegression

# Define LinearRegression algorithm
lrg_1 = LinearRegression(featuresCol = 'features', labelCol = 'output',regParam=2.0)
# maxIter: int = 100
model_1 = lrg_1.fit(trainingData)
predictions_1 = model_1.transform(testData)

predictions_1.show(5)

evaluator_1 = RegressionEvaluator(
    labelCol="output", predictionCol="prediction", metricName="rmse")
rmse_1 = evaluator_1.evaluate(predictions_1)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_1)
```

```
+------+--------------------+------------------+
|output|            features|        prediction|
+------+--------------------+------------------+
|  41.0|[6.53690364271165...|44.227559545991866|
|  44.0|[6.41599749220878...| 47.81024214005528|
|  44.0|[6.58719665904405...|50.707025368422734|
|  44.0|[6.60650222035666...|48.697781658265555|
|  45.0|[6.17380835592218...|48.883788636232325|
+------+--------------------+------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 2.14361
```

In [67]:

```python
from pyspark.ml.regression import LinearRegression

# Define LinearRegression algorithm
lrg_2 = LinearRegression(featuresCol = 'features', labelCol = 'output',maxIter=10)
model_2 = lrg_2.fit(trainingData)
predictions_2 = model_2.transform(testData)

predictions_2.show(5)

evaluator_2 = RegressionEvaluator(
    labelCol="output", predictionCol="prediction", metricName="rmse")
rmse_2 = evaluator_2.evaluate(predictions_2)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_2)
```

```
+------+--------------------+------------------+
|output|            features|        prediction|
+------+--------------------+------------------+
|  41.0|[6.53690364271165...| 41.53811522459333|
|  44.0|[6.41599749220878...| 45.92256921728185|
|  44.0|[6.58719665904405...| 48.79175549801562|
|  44.0|[6.60650222035666...|44.533694117718795|
|  45.0|[6.17380835592218...| 45.96323644678648|
+------+--------------------+------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 1.86664
```

In [68]:

```
from pyspark.ml.regression import LinearRegression

# Define LinearRegression algorithm
lrg_3 = LinearRegression(featuresCol = 'features', labelCol = 'output',maxIter=30)
model_3 = lrg_3.fit(trainingData)
predictions_3 = model_3.transform(testData)

predictions_3.show(5)

evaluator_3 = RegressionEvaluator(
    labelCol="output", predictionCol="prediction", metricName="rmse")
rmse_3 = evaluator_3.evaluate(predictions_3)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_3)
```

```
+------+--------------------+------------------+
|output|            features|        prediction|
+------+--------------------+------------------+
|  41.0|[6.53690364271165...| 40.53397568304155|
|  44.0|[6.41599749220878...| 45.15551922724518|
|  44.0|[6.58719665904405...|48.524183250218016|
|  44.0|[6.60650222035666...| 44.02133666015817|
|  45.0|[6.17380835592218...| 45.43579327618232|
+------+--------------------+------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 1.83285
```

In [69]:
```
from pyspark.ml.regression import LinearRegression

# Define LinearRegression algorithm
lrg_4 = LinearRegression(featuresCol = 'features', labelCol = 'output',maxIter=200)
model_4 = lrg_4.fit(trainingData)
predictions_4 = model_4.transform(testData)

predictions_4.show(5)

evaluator_4 = RegressionEvaluator(
    labelCol="output", predictionCol="prediction", metricName="rmse")
rmse_4 = evaluator_4.evaluate(predictions_4)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_4)
```

```
+------+--------------------+-----------------+
|output|            features|       prediction|
+------+--------------------+-----------------+
|  41.0|[6.53690364271165...|40.73535289532248|
|  44.0|[6.41599749220878...|45.15357021804499|
|  44.0|[6.58719665904405...|48.66151100088803|
|  44.0|[6.60650222035666...|  44.129994473203|
|  45.0|[6.17380835592218...|45.64501921754493|
+------+--------------------+-----------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 1.82613
```

In [ ]:
```

```

Tensorflow

In [70]:
```
from pyspark.sql.types import FloatType
```

In [71]:
```
to_array = udf(lambda v: v.toArray().tolist(), ArrayType(FloatType()))

df_test = testData
df_validate,df_train = trainingData.randomSplit([0.5,0.5])

df_train_pandas = df_train.withColumn('features', to_array('features')).toPandas()
df_validate_pandas = df_validate.withColumn('features', to_array('features')).toPandas()
df_test_pandas = df_test.withColumn('features', to_array('features')).toPandas()
```

In [72]:
```
import tensorflow as tf
from tensorflow import keras

x_train = tf.constant(np.array(df_train_pandas['features'].values.tolist()))
y_train = tf.constant(np.array(df_train_pandas['output'].values.tolist()))

x_validate = tf.constant(np.array(df_validate_pandas['features'].values.tolist()))
y_validate = tf.constant(np.array(df_validate_pandas['output'].values.tolist()))


x_test = tf.constant(np.array(df_test_pandas['features'].values.tolist()))
y_test = tf.constant(np.array(df_test_pandas['output'].values.tolist()))
```

In [73]:
```
print(x_train)
print(y_train)
```

```
tf.Tensor(
[[6.30738211e+00 7.97132349e+00 2.59242672e-03 ... 3.16481352e+00
  3.29432011e+00 7.40527344e+00]
 [6.22430420e+00 8.29017639e+00 3.45656904e-03 ... 6.43171787e+00
  3.48437715e+00 7.18747139e+00]
 [6.40443134e+00 8.60902977e+00 3.45656904e-03 ... 6.43171787e+00
  3.48437715e+00 7.18747139e+00]
 ...
 [6.02965415e-01 1.49860888e+01 1.50360746e+01 ... 9.29025936e+00
  2.09062624e+00 8.71208668e+00]
 [6.02965415e-01 1.49860888e+01 1.65051174e+01 ... 9.18816853e+00
  2.09062624e+00 8.71208668e+00]
 [4.58066463e+00 1.51455145e+01 1.91839581e+01 ... 9.69862270e+00
  1.52045548e+00 6.75186682e+00]], shape=(56176, 78), dtype=float64)
tf.Tensor([40. 42. 42. ... 94. 94. 94.], shape=(56176,), dtype=float64)
```

Neural networks

In [74]:
```
model_nn = keras.Sequential( [keras.layers.Dense(78,activation='relu'),
                              keras.layers.Dense(10,activation='relu'),
                              keras.layers.Dense(10,activation='relu'),
                              keras.layers.Dense(10,activation='relu') ,
                              keras.layers.Dense(1)] )

```

In [75]:
```
y_pred = model_nn(x_train)
model_nn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (56176, 78)               6162

 dense_1 (Dense)             (56176, 10)               790

 dense_2 (Dense)             (56176, 10)               110

 dense_3 (Dense)             (56176, 10)               110

 dense_4 (Dense)             (56176, 1)                11

=================================================================
Total params: 7,183
Trainable params: 7,183
Non-trainable params: 0
_________________________________________________________________
```

In [76]:
```
print(y_pred)
print(y_train)
```

```
tf.Tensor(
[[74.732086]
 [74.12477 ]
 [74.16112 ]
 ...
 [73.94921 ]
 [74.17415 ]
 [74.34247 ]], shape=(56176, 1), dtype=float32)
tf.Tensor([40. 42. 42. ... 94. 94. 94.], shape=(56176,), dtype=float64)
```

In [77]:
```
mse = keras.losses.MeanSquaredError()

model_nn.compile(optimizer = 'adam',
    loss=mse,
    metrics=[mse])
model_nn.fit(x_train,y_train, epochs = 20,validation_data=(x_validate,y_validate),verbose = 2)

loss = mse(y_train, y_pred).numpy()
print(loss)
```

```
s/step
Epoch 11/20
1756/1756 - 5s - loss: 2.8021 - mean_squared_error: 2.8017 - val_loss: 3.2102 - val_mean_squared_error: 3.2180 - 5s/epoch - 3m
s/step
Epoch 12/20
1756/1756 - 5s - loss: 2.6239 - mean_squared_error: 2.6245 - val_loss: 2.4021 - val_mean_squared_error: 2.4094 - 5s/epoch - 3m
s/step
Epoch 13/20
1756/1756 - 5s - loss: 2.5039 - mean_squared_error: 2.5045 - val_loss: 4.0864 - val_mean_squared_error: 4.0895 - 5s/epoch - 3m
s/step
Epoch 14/20
1756/1756 - 5s - loss: 2.3757 - mean_squared_error: 2.3755 - val_loss: 2.2760 - val_mean_squared_error: 2.2826 - 5s/epoch - 3m
s/step
Epoch 15/20
1756/1756 - 4s - loss: 2.2276 - mean_squared_error: 2.2274 - val_loss: 2.0466 - val_mean_squared_error: 2.0521 - 4s/epoch - 3m
s/step
Epoch 16/20
1756/1756 - 5s - loss: 2.1436 - mean_squared_error: 2.1435 - val_loss: 1.8553 - val_mean_squared_error: 1.8600 - 5s/epoch - 3m
s/step
Epoch 17/20
```

In [78]:
```
model_nn.evaluate(x_test,y_test, verbose = 2)
```

```
869/869 - 1s - loss: 1.6491 - mean_squared_error: 1.6550 - 985ms/epoch - 1ms/step
```

Out[78]:
```
[1.6490752696990967, 1.6549885272979736]
```

In [ ]:
```

```

In [ ]:
```

```

In [79]:
```
def cross_valiation(hyper,k,s_r,x,y,logdir):

    def data_split():
        for i in range(k):
            idx=tf.range(df_train_pandas.shape[0])
            splt_idx = int(s_r*df_train_pandas.shape[0])
            idx = tf.random.shuffle(idx)
            x_train, y_train = tf.gather(x, idx[:splt_idx]), tf.gather(y, idx[:splt_idx])
            x_valid, y_valid = tf.gather(x, idx[splt_idx:]), tf.gather(y, idx[splt_idx:])
            return x_train,y_train,x_valid,y_valid

    model = keras.Sequential()
    for _ in range(hparams[HP_DEPTH]):
        model.add(keras.layers.Dense(hparams[HP_WIDTH],activation='relu'))
        model.add(keras.layers.Dense(1))
        model.compile(optimizer = 'adam',
            loss=keras.losses.MeanSquaredError(),
            metrics=[keras.losses.MeanSquaredError(name = 'MSE')])
        x_train,y_train,x_valid,y_valid = data_split()
        history = model.fit(x_train, y_train, epochs=5, verbose = 2,validation_data = (x_valid, y_valid),
                  callbacks=[tf.keras.callbacks.TensorBoard(log_dir=logdir, histogram_freq=1)])
        accuracy = np.mean(history.history["MSE"])
        model.summary()
    return accuracy
```

In [80]:

```
from tensorboard.plugins.hparams import api as hp

HP_WIDTH = hp.HParam('NN_width', hp.Discrete([10,20,30]))
HP_DEPTH = hp.HParam('NN_depth', hp.Discrete([3,4,5]))


with tf.summary.create_file_writer('logs14813/hparam_tuning').as_default():
    hp.hparams_config(
    hparams=[HP_WIDTH, HP_DEPTH],
    metrics=[hp.Metric('MSE')],
  )

```

In [81]:

```
import datetime
for hp_width in HP_WIDTH.domain.values:
    for hp_depth in (HP_DEPTH.domain.values):
        hparams = {
            HP_WIDTH: hp_width,
            HP_DEPTH: hp_depth,
        }
        run_name = f"run-WIDTH{int(hparams[HP_WIDTH])}-DEPTH{hparams[HP_DEPTH]}"
        print('--- Starting trial: %s' % run_name)
        print({h.name: hparams[h] for h in hparams})

        run_dir = 'logs14813/hparam_tuning/' + datetime.datetime.now().strftime("%Y%m%d-%H%M%S") + run_name
        accuracy = cross_valiation(hparams,10,0.7,x_train, y_train,run_dir)

        with tf.summary.create_file_writer(run_dir).as_default():
          hp.hparams(hparams)  # record the values used in this trial
          tf.summary.scalar("MSE", accuracy, step=1)
```

```
dense_70 (Dense)             (None, 1)                 31

dense_71 (Dense)             (None, 30)                60

dense_72 (Dense)             (None, 1)                 31

dense_73 (Dense)             (None, 30)                60

dense_74 (Dense)             (None, 1)                 31

dense_75 (Dense)             (None, 30)                60

dense_76 (Dense)             (None, 1)                 31

=================================================================
Total params: 2,765
Trainable params: 2,765
Non-trainable params: 0
_________________________________________________________________
```

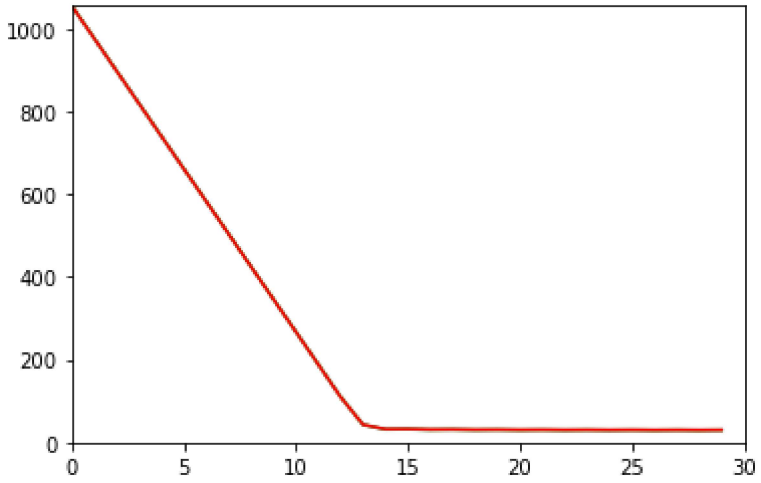
Linear Regression

In [82]:

```
tf.compat.v1.disable_eager_execution()
# tf.compat.v1.enable_eager_execution()
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

from sklearn.datasets import load_boston

def append_bias_reshape(x_train,y_train):
    n_training_samples = x_train.shape[0]
    n_dim = x_train.shape[1]
    f = np.reshape(np.c_[np.ones(n_training_samples),x_train],[n_training_samples,n_dim+1])
    l = np.reshape(y_train,[n_training_samples,1])

    return f,l

if __name__ == '__main__':


#     x,y = read_boston_data()
#     norm_features = feature_normalize(x)
    f,l = append_bias_reshape(x_train,y_train)
    n_dim = f.shape[1]

    rnd_indices = np.random.rand(len(f)) < 0.80

    x_train = f[rnd_indices]
    y_train = l[rnd_indices]
    x_test = f[~rnd_indices]
    y_test = l[~rnd_indices]



    learning_rate = 0.0001
    training_epochs = 30
    cost_history = []
    test_history = []

    X = tf.compat.v1.placeholder(tf.float32,[None,n_dim])
    Y = tf.compat.v1.placeholder(tf.float32,[None,1])
    W = tf.Variable(tf.ones([n_dim,1]))

    init = tf.compat.v1.initialize_all_variables()
```

```
44
45      y_ = tf.matmul(X,W)
46      cost = tf.reduce_mean(tf.abs(y_-Y))
47      training_step = tf.compat.v1.train.GradientDescentOptimizer(learning_rate).minimize(cost)
48
49      sess = tf.compat.v1.Session()
50      sess.run(init)
51
52      for epoch in range(training_epochs):
53          sess.run(training_step,feed_dict={X:x_train,Y:y_train})
54          c = sess.run(cost,feed_dict={X:x_train,Y:y_train})
55          print(c)
56          t = sess.run(cost,feed_dict={X:x_test,Y:y_test})
57          print(t)
58          cost_history.append(c)
59          test_history.append(t)
60
61      plt.plot(range(len(test_history)),test_history,color = 'green')
62      plt.plot(range(len(cost_history)),cost_history,color = 'red')
63
64      plt.axis([0,training_epochs,0,np.max(cost_history)])
65      plt.show()
```

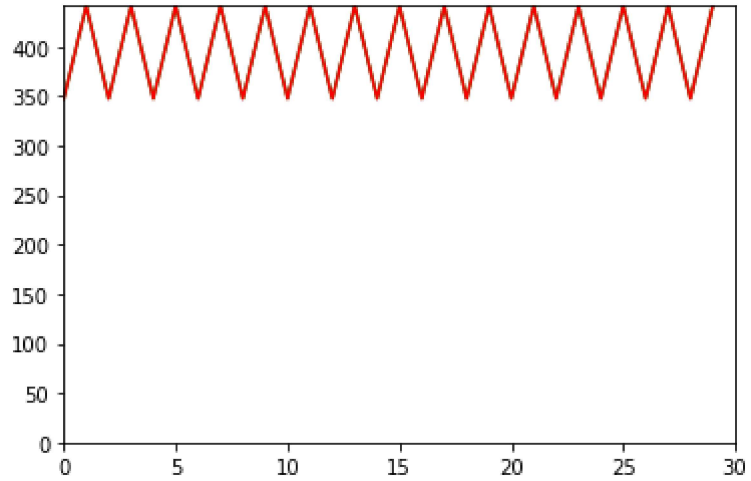
```
30.006186
29.442467
30.151493
29.889187
```



In [86]:

```
# tf.compat.v1.disable_eager_execution()
# tf.compat.v1.enable_eager_execution()
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

from sklearn.datasets import load_boston

def append_bias_reshape(x_train,y_train):
    n_training_samples = x_train.shape[0]
    n_dim = x_train.shape[1]
    f = np.reshape(np.c_[np.ones(n_training_samples),x_train],[n_training_samples,n_dim+1])
    l = np.reshape(y_train,[n_training_samples,1])

    return f,l

if __name__ == '__main__':


#     x,y = read_boston_data()
#     norm_features = feature_normalize(x)
    f,l = append_bias_reshape(x_train,y_train)
    n_dim = f.shape[1]

    rnd_indices = np.random.rand(len(f)) < 0.80

    x_train = f[rnd_indices]
    y_train = l[rnd_indices]
    x_test = f[~rnd_indices]
    y_test = l[~rnd_indices]



    learning_rate = 0.001
    training_epochs = 30
    cost_history = []
    test_history = []

    X = tf.compat.v1.placeholder(tf.float32,[None,n_dim])
    Y = tf.compat.v1.placeholder(tf.float32,[None,1])
    W = tf.Variable(tf.ones([n_dim,1]))

    init = tf.compat.v1.initialize_all_variables()
```

```
44
45      y_ = tf.matmul(X,W)
46      cost = tf.reduce_mean(tf.abs(y_-Y))
47      training_step = tf.compat.v1.train.GradientDescentOptimizer(learning_rate).minimize(cost)
48
49      sess = tf.compat.v1.Session()
50      sess.run(init)
51
52      for epoch in range(training_epochs):
53          sess.run(training_step,feed_dict={X:x_train,Y:y_train})
54          c = sess.run(cost,feed_dict={X:x_train,Y:y_train})
55          print(c)
56          t = sess.run(cost,feed_dict={X:x_test,Y:y_test})
57          print(t)
58          cost_history.append(c)
59          test_history.append(t)
60
61      plt.plot(range(len(test_history)),test_history,color = 'green')
62      plt.plot(range(len(cost_history)),cost_history,color = 'red')
63
64      plt.axis([0,training_epochs,0,np.max(cost_history)])
65      plt.show()
```

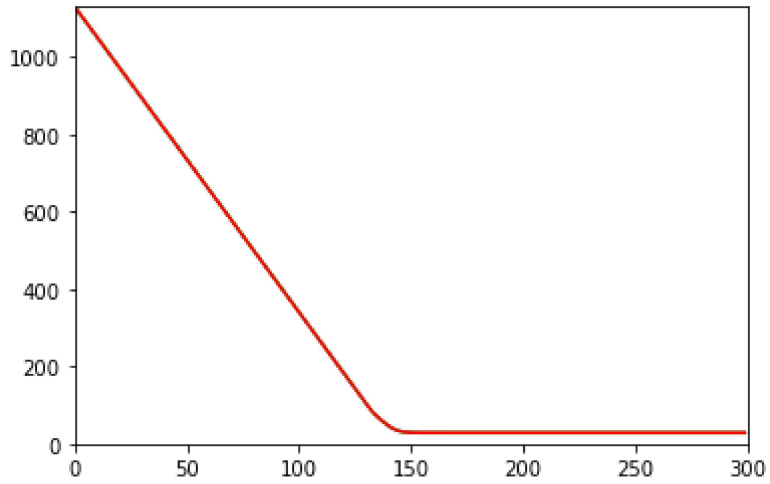


In [87]:

```
# tf.compat.v1.disable_eager_execution()
# tf.compat.v1.enable_eager_execution()
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

from sklearn.datasets import load_boston

def append_bias_reshape(x_train,y_train):
    n_training_samples = x_train.shape[0]
    n_dim = x_train.shape[1]
    f = np.reshape(np.c_[np.ones(n_training_samples),x_train],[n_training_samples,n_dim+1])
    l = np.reshape(y_train,[n_training_samples,1])

    return f,l

if __name__ == '__main__':


#     x,y = read_boston_data()
#     norm_features = feature_normalize(x)
    f,l = append_bias_reshape(x_train,y_train)
    n_dim = f.shape[1]

    rnd_indices = np.random.rand(len(f)) < 0.80

    x_train = f[rnd_indices]
    y_train = l[rnd_indices]
    x_test = f[~rnd_indices]
    y_test = l[~rnd_indices]



    learning_rate = 0.00001
    training_epochs = 300
    cost_history = []
    test_history = []

    X = tf.compat.v1.placeholder(tf.float32,[None,n_dim])
    Y = tf.compat.v1.placeholder(tf.float32,[None,1])
    W = tf.Variable(tf.ones([n_dim,1]))

    init = tf.compat.v1.initialize_all_variables()
```

```
44
45      y_ = tf.matmul(X,W)
46      cost = tf.reduce_mean(tf.abs(y_-Y))
47      training_step = tf.compat.v1.train.GradientDescentOptimizer(learning_rate).minimize(cost)
48
49      sess = tf.compat.v1.Session()
50      sess.run(init)
51
52      for epoch in range(training_epochs):
53          sess.run(training_step,feed_dict={X:x_train,Y:y_train})
54          c = sess.run(cost,feed_dict={X:x_train,Y:y_train})
55          print(c)
56          t = sess.run(cost,feed_dict={X:x_test,Y:y_test})
57          print(t)
58          cost_history.append(c)
59          test_history.append(t)
60
61      plt.plot(range(len(test_history)),test_history,color = 'green')
62      plt.plot(range(len(cost_history)),cost_history,color = 'red')
63
64      plt.axis([0,training_epochs,0,np.max(cost_history)])
65      plt.show()
```

```
28.475595
28.635077
28.475382
28.634863
28.475172
```

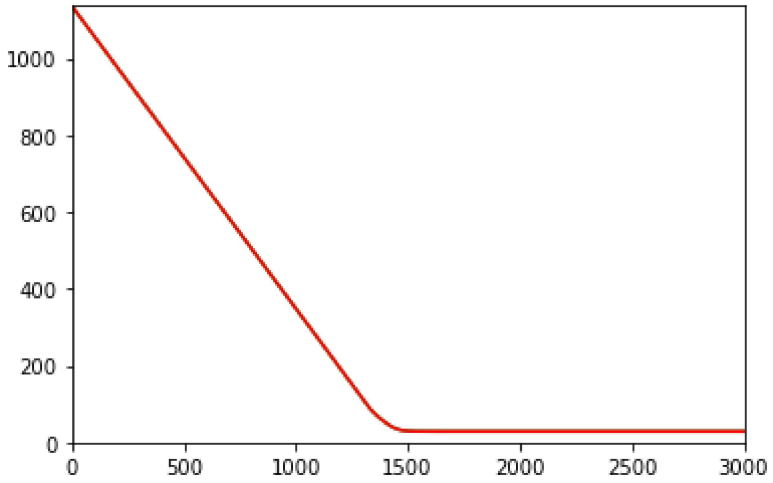


In [88]:

```
# tf.compat.v1.disable_eager_execution()
# tf.compat.v1.enable_eager_execution()
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

from sklearn.datasets import load_boston

def append_bias_reshape(x_train,y_train):
    n_training_samples = x_train.shape[0]
    n_dim = x_train.shape[1]
    f = np.reshape(np.c_[np.ones(n_training_samples),x_train],[n_training_samples,n_dim+1])
    l = np.reshape(y_train,[n_training_samples,1])

    return f,l

if __name__ == '__main__':


#     x,y = read_boston_data()
#     norm_features = feature_normalize(x)
    f,l = append_bias_reshape(x_train,y_train)
    n_dim = f.shape[1]

    rnd_indices = np.random.rand(len(f)) < 0.80

    x_train = f[rnd_indices]
    y_train = l[rnd_indices]
    x_test = f[~rnd_indices]
    y_test = l[~rnd_indices]



    learning_rate = 0.000001
    training_epochs = 3000
    cost_history = []
    test_history = []

    X = tf.compat.v1.placeholder(tf.float32,[None,n_dim])
    Y = tf.compat.v1.placeholder(tf.float32,[None,1])
    W = tf.Variable(tf.ones([n_dim,1]))

    init = tf.compat.v1.initialize_all_variables()
```

```
44
45      y_ = tf.matmul(X,W)
46      cost = tf.reduce_mean(tf.abs(y_-Y))
47      training_step = tf.compat.v1.train.GradientDescentOptimizer(learning_rate).minimize(cost)
48
49      sess = tf.compat.v1.Session()
50      sess.run(init)
51
52      for epoch in range(training_epochs):
53          sess.run(training_step,feed_dict={X:x_train,Y:y_train})
54          c = sess.run(cost,feed_dict={X:x_train,Y:y_train})
55          print(c)
56          t = sess.run(cost,feed_dict={X:x_test,Y:y_test})
57          print(t)
58          cost_history.append(c)
59          test_history.append(t)
60
61      plt.plot(range(len(test_history)),test_history,color = 'green')
62      plt.plot(range(len(cost_history)),cost_history,color = 'red')
63
64      plt.axis([0,training_epochs,0,np.max(cost_history)])
65      plt.show()
```

```
28.593613
28.796995
28.593592
28.796976
```



In [ ]:

```
1
```