Big Data Analytics and Data Visualisation

Harris Boyle

Analysis of Football Manager™ player data to predict individual performance, value, and compare to their real life counter parts

Introduction

This project aims to analyse game data from Football Manager 2020™ to gather some interesting and predictive outcomes that demonstrate the usage and applications of PySpark and Tableau with respect to big data problems.

Whilst the data set is not truly “big data” it is a very large data set comprising some 144,750 rows and 64 columns. Comprising of several numerical, categorical and string data types.

A small extract from Tableau is provided below to give the reader a brief overview of what the data looks like.

datafm20.csv 71 fields 144750 rows 100 rows

Table Details

	F1	Name	Position	Club	Division	Based	Nation	Height	Weight	Age	Preferred Foot	Best Pos	Best Role	Value
0	0	Lionel Messi	AM (RC), ST (C)	Barcelona	Spanish First Division	Spain (First Division)	ARG	170	72	32	Left	AM (R)	IF	73.00
1	1	Cristiano Ronaldo	AM (RL), ST (C)	Juventus	Italian Serie A	Italy (Serie A)	POR	185	83	34	Either	ST (C)	CF	31.00
2	2	Kylian MbappÃ©	AM (RL), ST (C)	Paris SG	Ligue 1 Conforama	France (Ligue 1 Conforama)	FRA	178	73	20	Right	ST (C)	AF	86.00
3	3	Manuel Neuer	GK	FC Bayern	Bundesliga	Germany (Bundesliga)	GER	192	90	33	Either	GK	SK	44.00
4	4	Neymar	M (L), AM (LC), ST (C)	Paris SG	Ligue 1 Conforama	France (Ligue 1 Conforama)	BRA	175	68	27	Right	AM (L)	IW	91.00
5	5	Erling Haaland	ST (C)	Borussia Dortmund	Bundesliga	Germany (Bundesliga)	NOR	194	87	18	Left	ST (C)	AF	67.00
6	6	Kevin De Bruyne	M (RLC), AM (C)	Man City	English Premier Division	England (Premier Division)	BEL	181	70	27	Either	M (C)	MEZ	88.00

Analysis of the data set is being performed using PySpark within Jupyter Notebook (a python IDE distributed with Anaconda), and, with use of Tableau for data visualisation and exploration.

The analysis will comprise of attempts to predict some label from other variables within the data set, and to predict some value from other variables also. Within the report player “Value” has been predicted using all other variables. Additionally, “Based” (where a player is based), “Nation” and “Best Pos” (a players best position) have been predicted using all other variables.

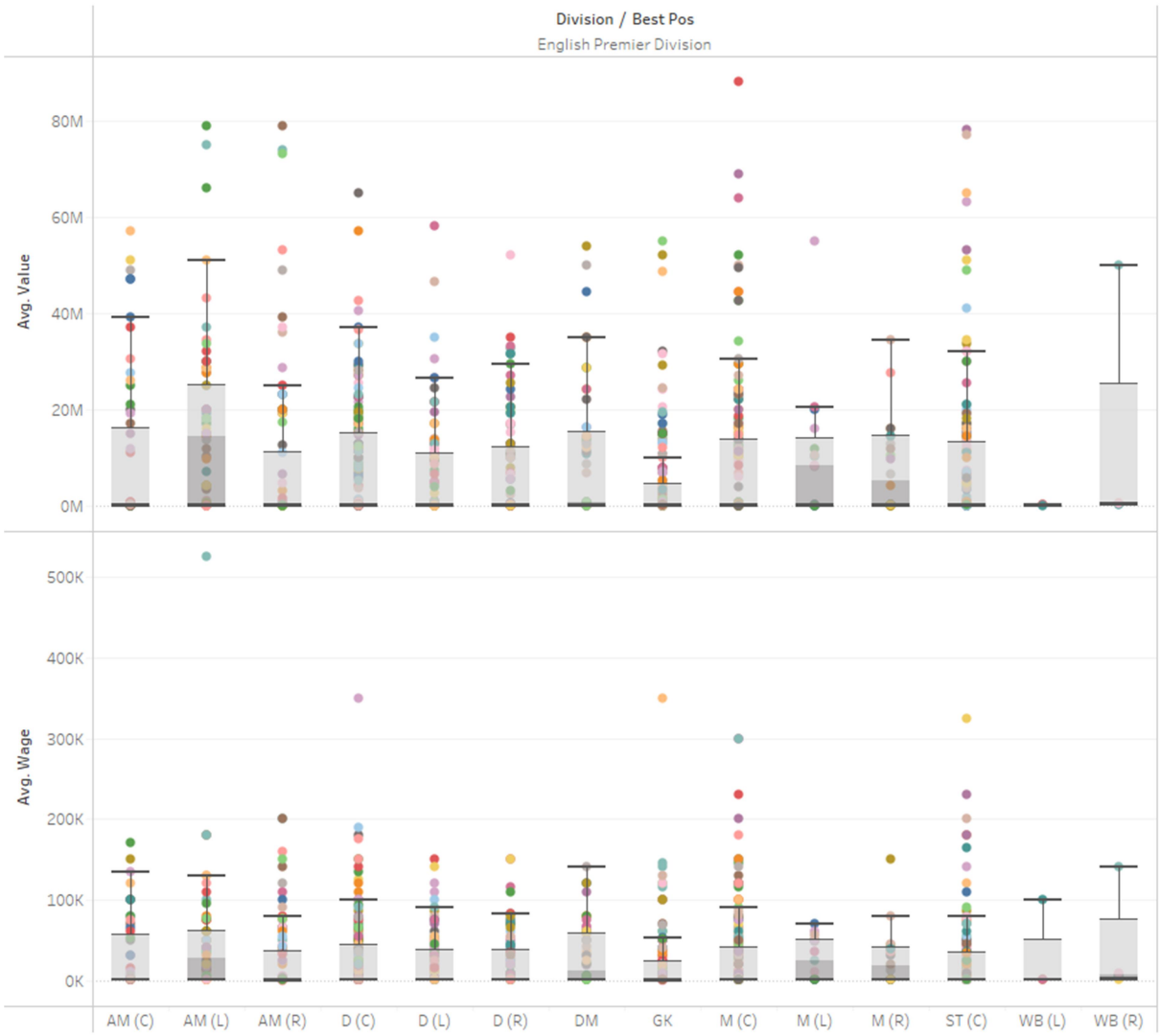
Regression modelling is to be used for the “Value” predictions and several times of machine learning classification are used for the others. Namely: Decision Tree, Logistic Regression and Random Forest.

In order for them to be easily repeatable and many different models attempted without much additional coding, Pipelines have been used within PySpark to achieve a streamlined and rapid approach once data cleansing/wrangling has been completed.

Exploratory Analysis

Exploratory analysis was undertaken in Tableau before any data cleansing/wrangling was completed. This allowed for easy visual spotting of what may or may not work, and what may or may not need cleansing prior to more detailed analysis and predictive analytics commencing.

Box Plots Prem

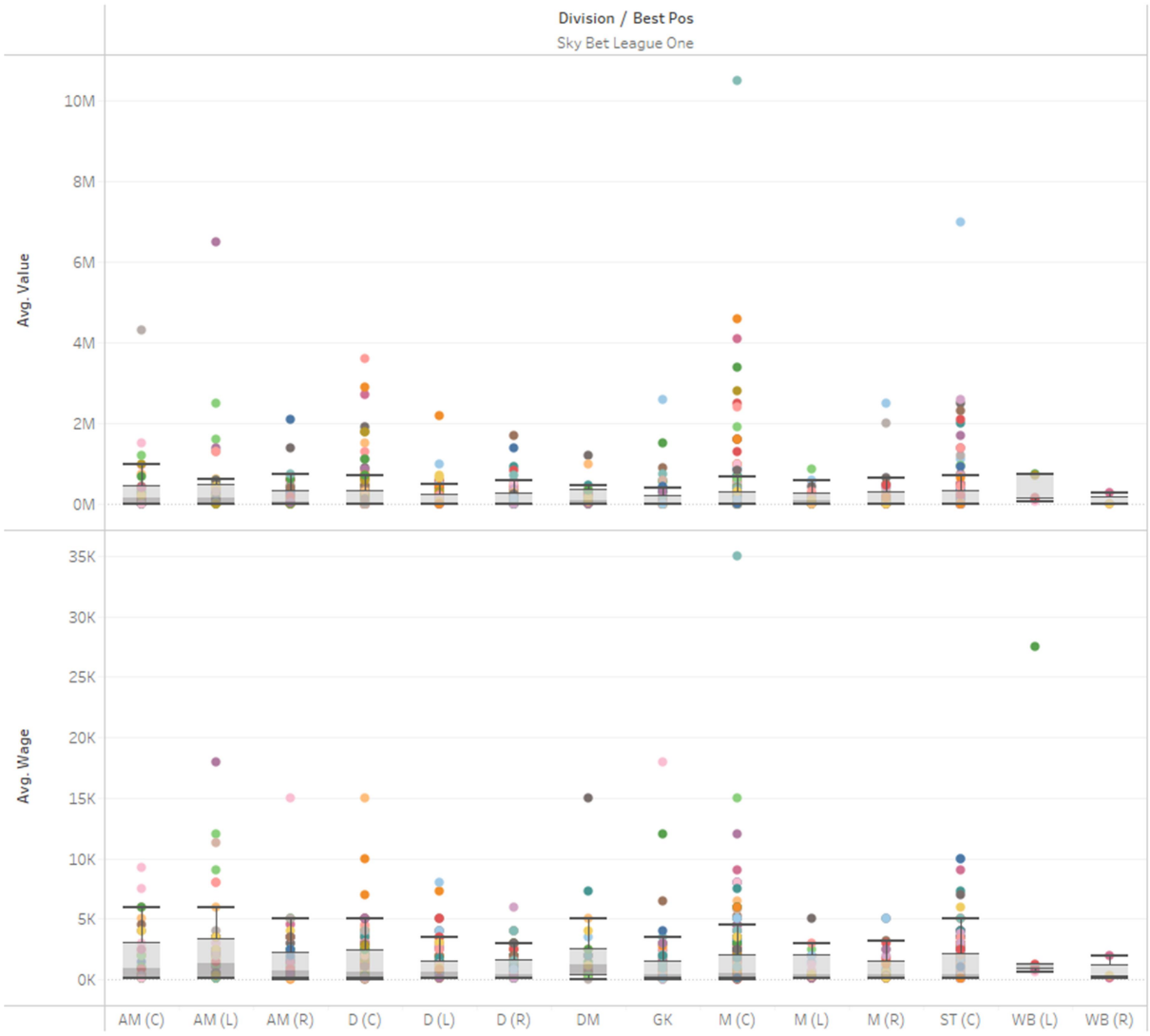


Average of Valueandaverage of Wage for each Best Pos broken down by Division. Colour shows details about Name. Details are shown for Preferred FootandClub. For pane Average of Wage: Details are shown for Preferred Foot,Best PosandClub.The view is filtered on Division, which keeps English Premier Division.

We can very quickly see that accurately predicting Wage or Value is likely to be difficult, there are a great deal of outliers within a single division only even when split by position, and, though not included here, when division is not filtered the matter is far worse.

For comparison I also produced plots for the lower division of English Football and for the German top flight (the Bundesliga). Both can be seen below respectively.

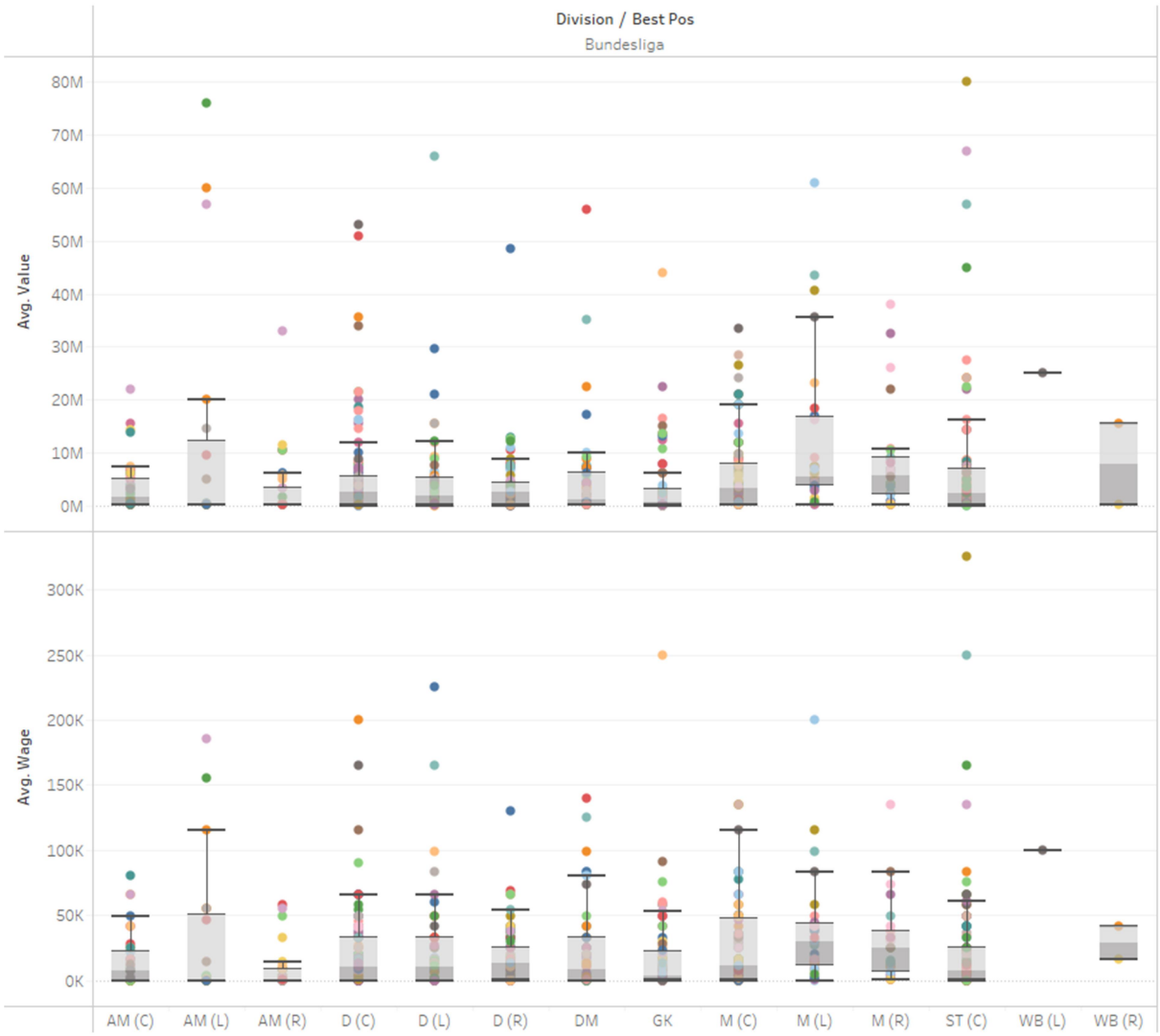
Box Plots L1



Average of Valueandaverage of Wage for each Best Pos broken down by Division. Colour shows details about Name. Details are shown for Preferred FootandClub. For pane Average of Value: Details are shown for Preferred Foot,Best PosandClub.The view is filtered on Division, which keeps Sky Bet League One.

We very quickly notice that English League One is very different to the Premiership. Whilst again, there are many outliers to consider, there is additionally a huge change in the scale. If we were to produce some predictive analysis of player value or wage, we may want to restrict is heavily to certain clusters of players. Perhaps the model for League One would look very different to the Premiership for example. It appears very unlikely that we would find some model that accurate fits every player without some segmentation.

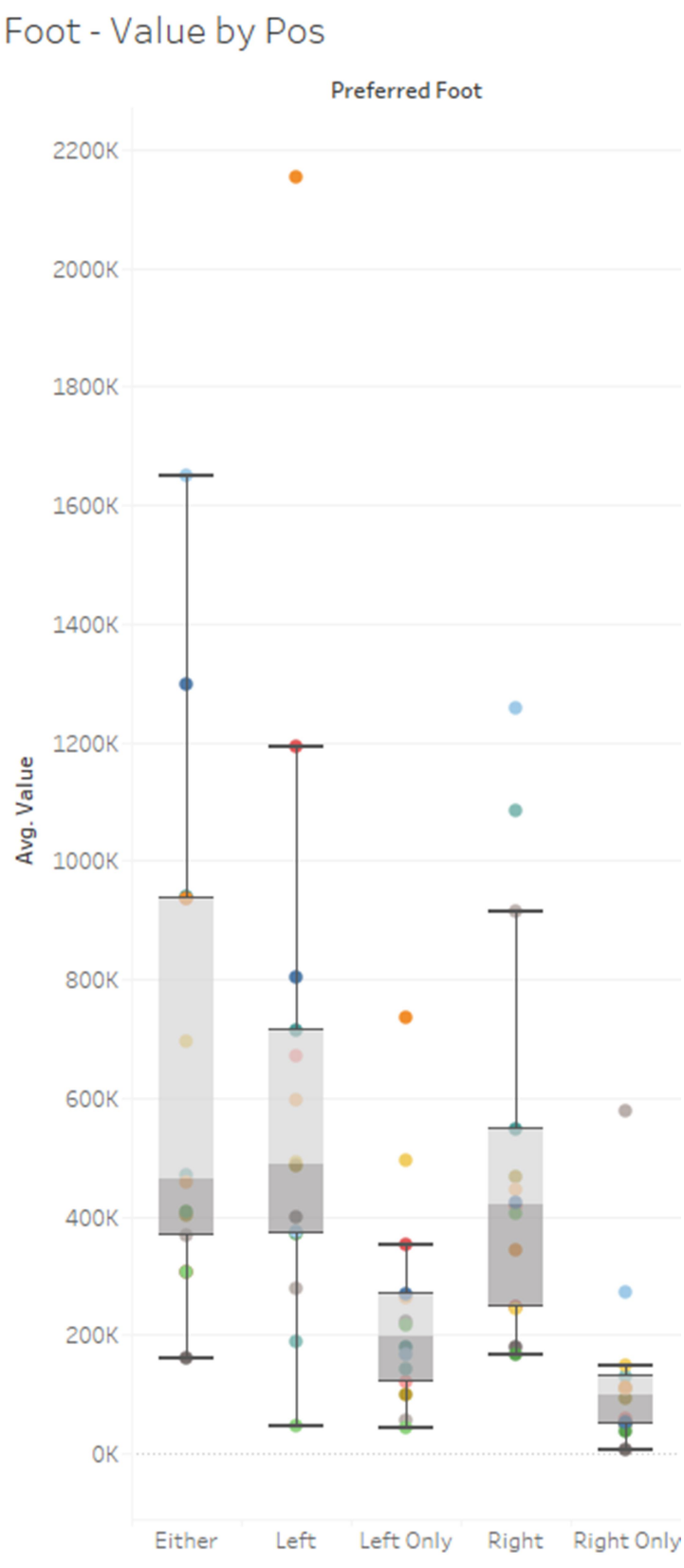
Box Plots Bundesliga



Average of Valueandaverage of Wage for each Best Pos broken down by Division. Colour shows details about Name. Details are shown for Preferred FootandClub. For pane Average of Value: Details are shown for Preferred Foot,Best PosandClub.The view is filtered on Division, which keeps Bundesliga.

As another top flight division, the Bundesliga much more closely mirrors the Premiership. This would lend further credence to the possibility that we would possibly want a separate model for each division when predicting salaries or perhaps, that we should be excluding Division all together. It may be that the reverse is a better predictor, i.e. how much money you make may better determine the division you play in rather than the reverse. A players division is likely some combination of their attributes as a player, and thus, being more skilled affords them entry into a higher division and the higher salary that comes with it. Principal Component Analysis may be useful here as we will likely have some conflating features which could benefit from reduction / combination.

Further down the vein of player value, preferred foot could have some impact on the value of a player. Two footed players generally being considered more skilled and thus potentially more desirable.

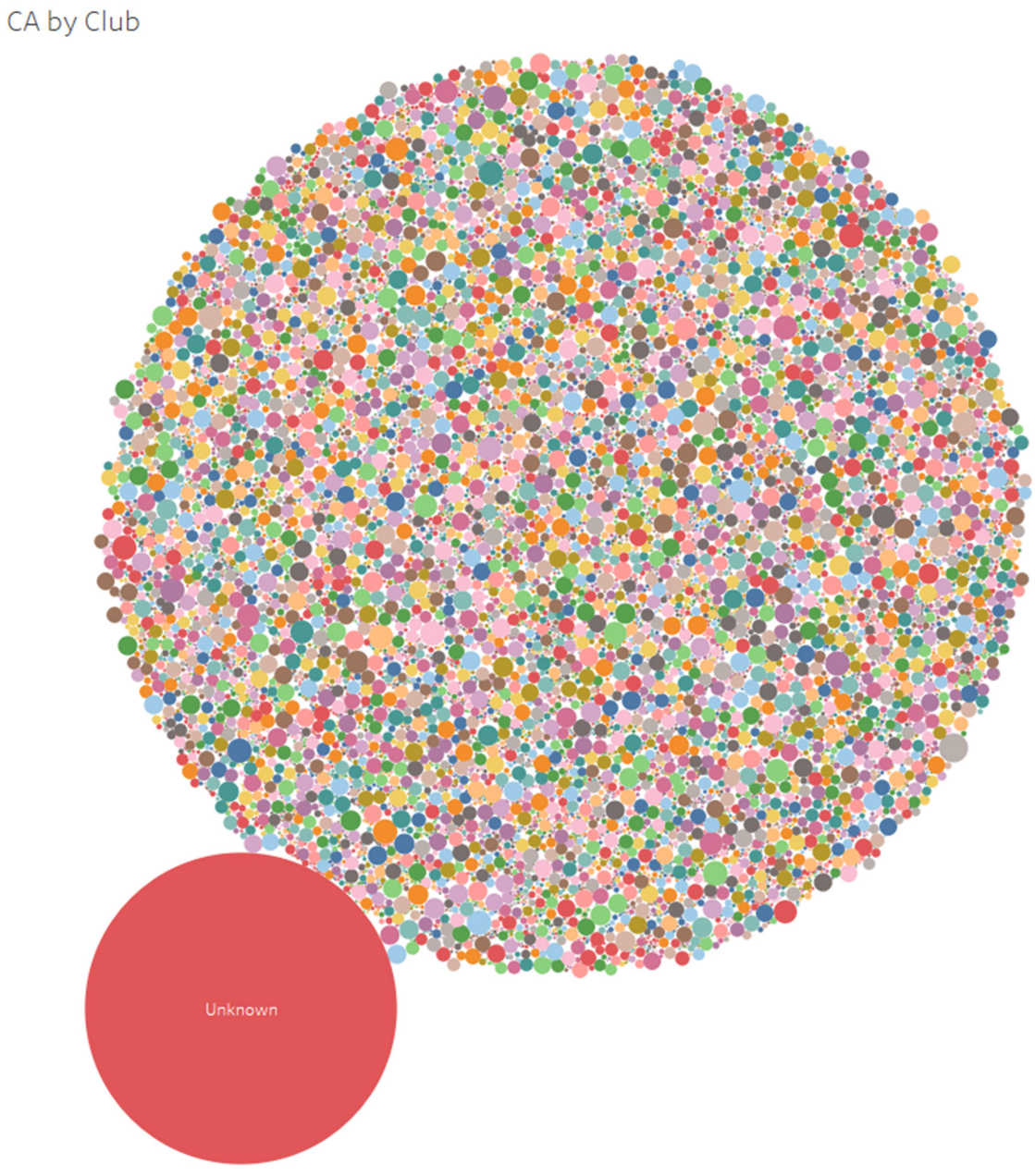


Average of Value for each Preferred Foot. Colour shows details about Best Pos.

This hypothesis is upheld in the box plot above, generally speaking two footed players have a higher value on average. However, the large outliers on Left and Right are with respect to

their opposite wing. For example, a Right footed Left winger and a Left footed Right winger. This also makes sense, generally speaking this makes you a better player when attacking in the position due to the facing of the opposition. Likewise, the very bottom of the whiskers on Left and Right are with respect to Defenders of the opposite foot as generally this makes you weaker in the position. From this plot we get some indication that your preferred foot may have a considerable impact on your value, particularly when considering your position also.

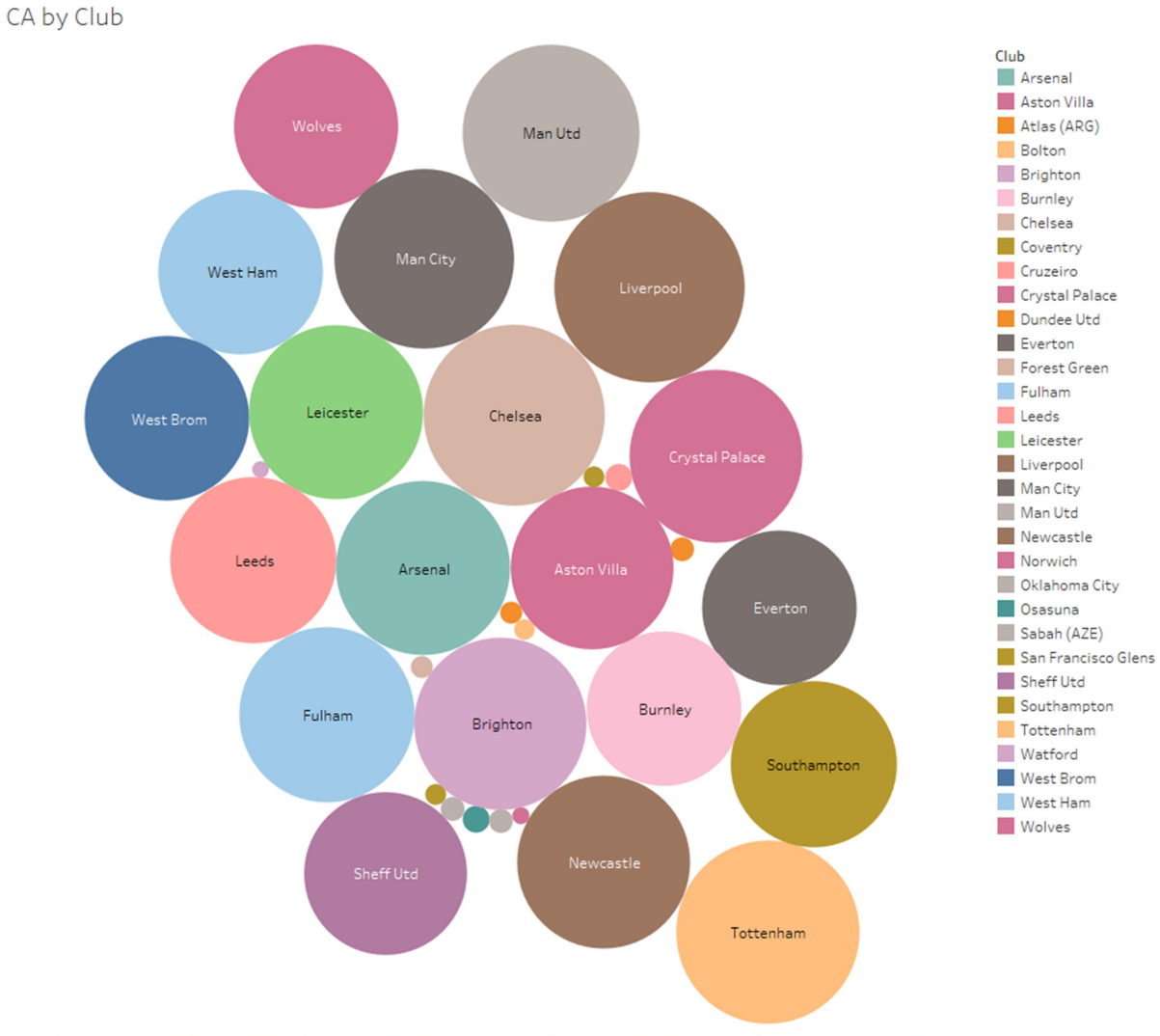
Current Ability and Potential Ability are some combination of players other perceived attributes, one thing I expected to see was that the combined ability of all players at a club would be higher in the better teams than in the worse teams. However, the first plot I produced showed a larger issue.



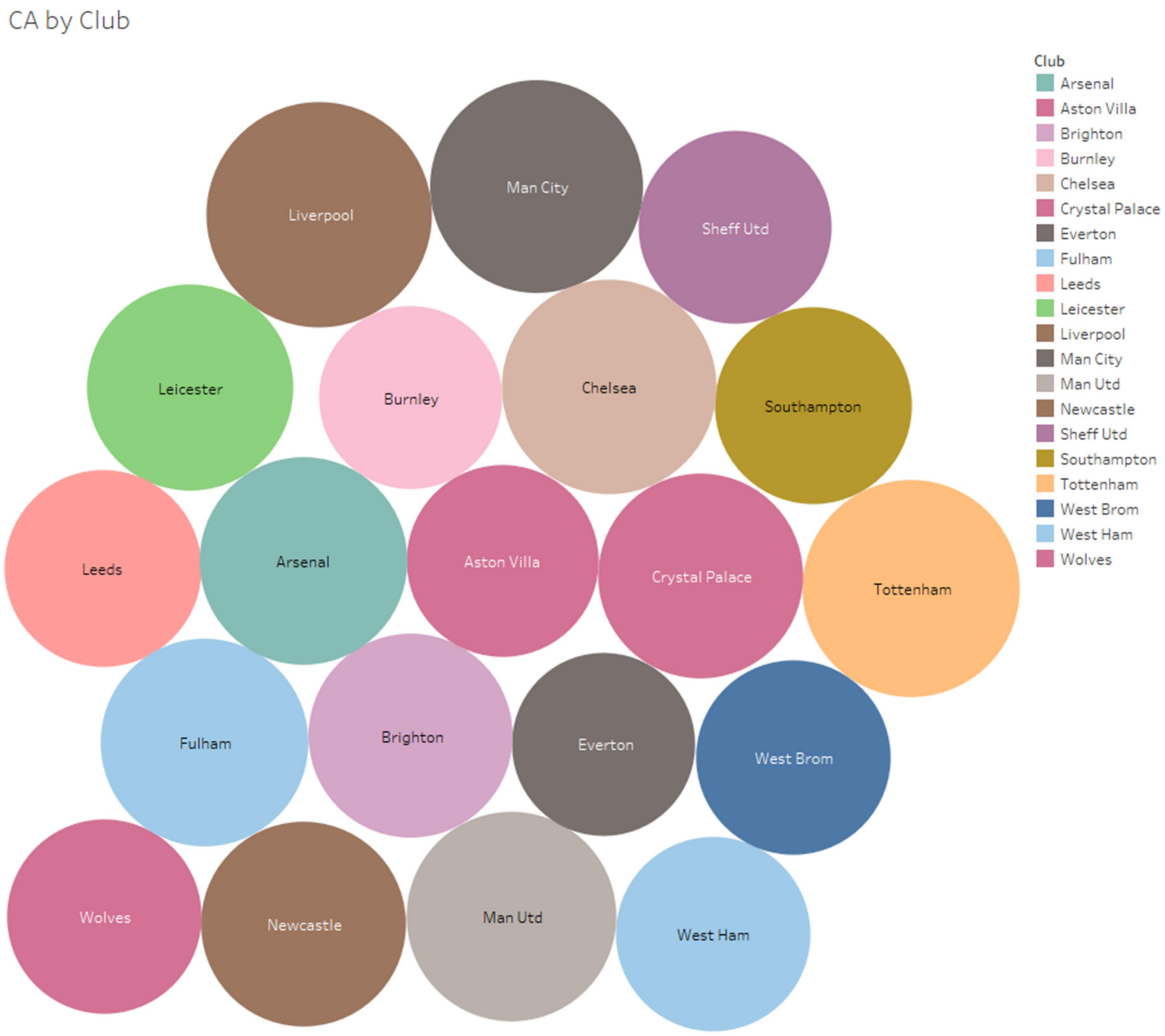
We can very easily see from the plot that we have a huge amount of unknowns which will need to be removed during our analysis later.

```
In [34]: #Remove players with "Unknown" clubs from the data set
         fm_data_filtered = fm_data.filter(fm_data['Club']!="Unknown")

         unknowns = Original - fm_data_filtered.count()
         print("Unknowns removed: ", unknowns)

          Unknowns removed:  20717
```

In the end a useless plot for analysis, but it provided some very quick insight into a data cleansing issue.

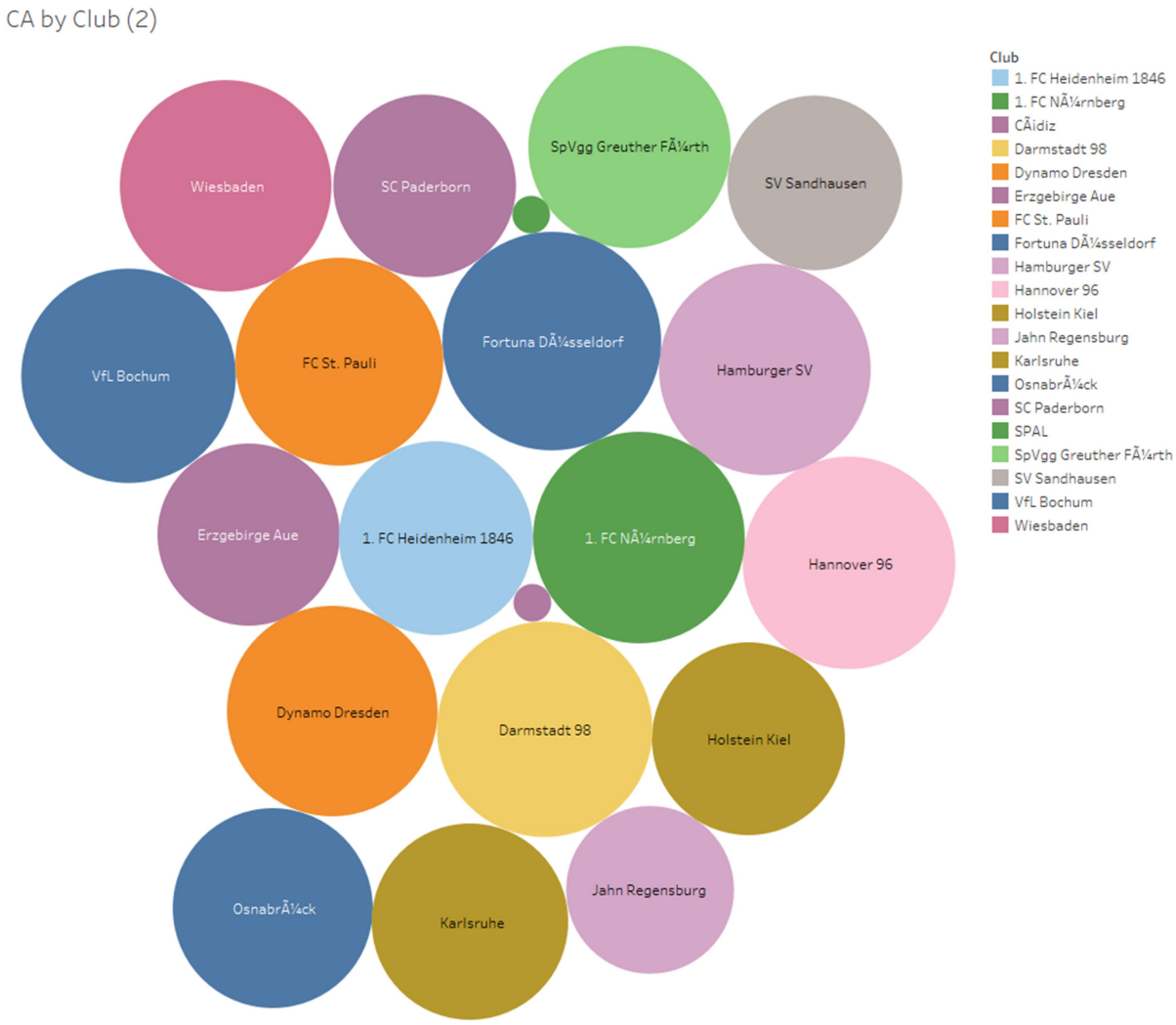


Filtering the same plot for just the Premiership flagged another data issue, some clubs are not correctly allocated to their division. However, in the interest of time and expedience no cleansing has been performed on this given the sheer size of the data set and the labour required to correctly allocate all clubs. However, a simple replace can be conducted using PySpark (and is implemented for a different data cleansing issue later). Given more time, this would be a good thing to revisit.



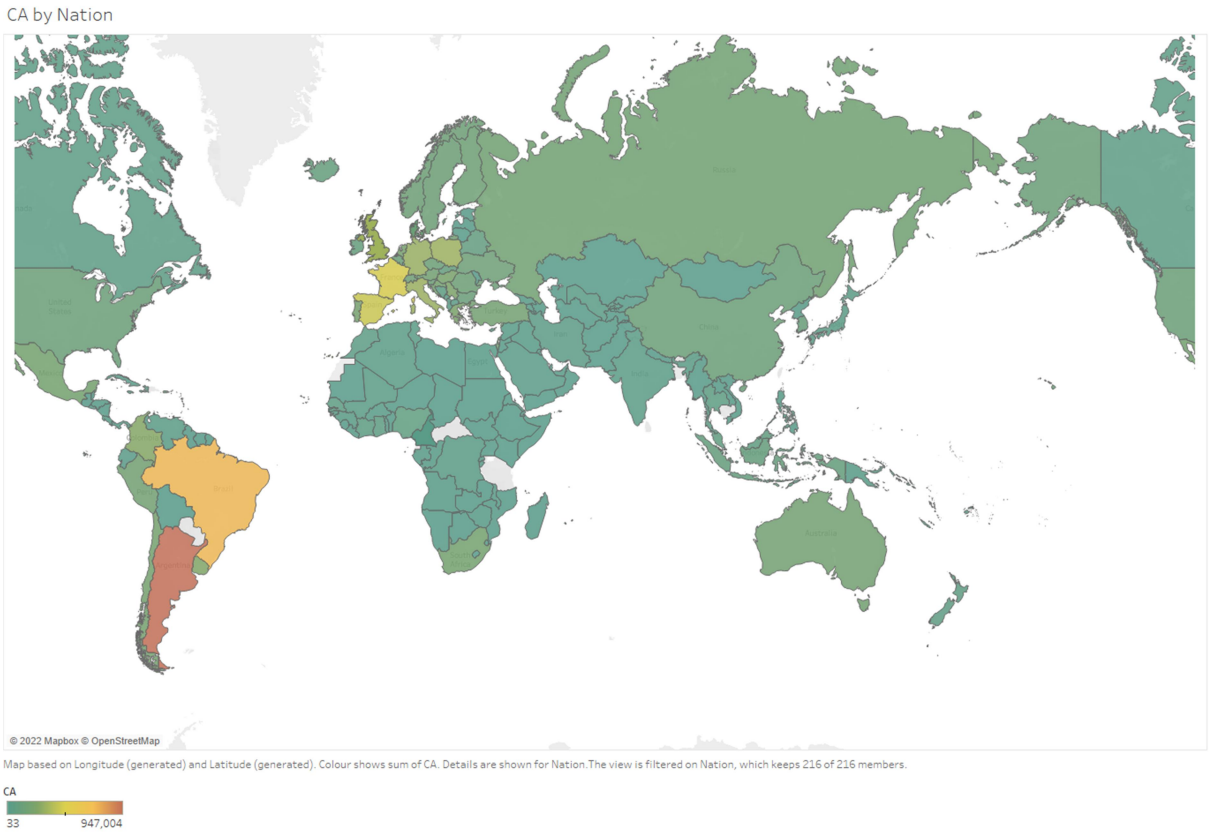
Club. Colour shows details about Club. Size shows sum of CA. The marks are labelled by Club. Details are shown for Division.The view is filtered on DivisionandExclusions (Club,Division).The Division filter keeps English Premier Division.The Exclusions (Club,Division) filter keeps 14,758 members.

Once these are manually filtered out, we see that, generally the Premiership is a closely skilled division (which mirrors real life given the generally accepted adage that the Premiership is the most competitive league in the world). Comparatively the second division of the German league (Bundesliga 2) shows a much larger disparity between teams as we would expect.



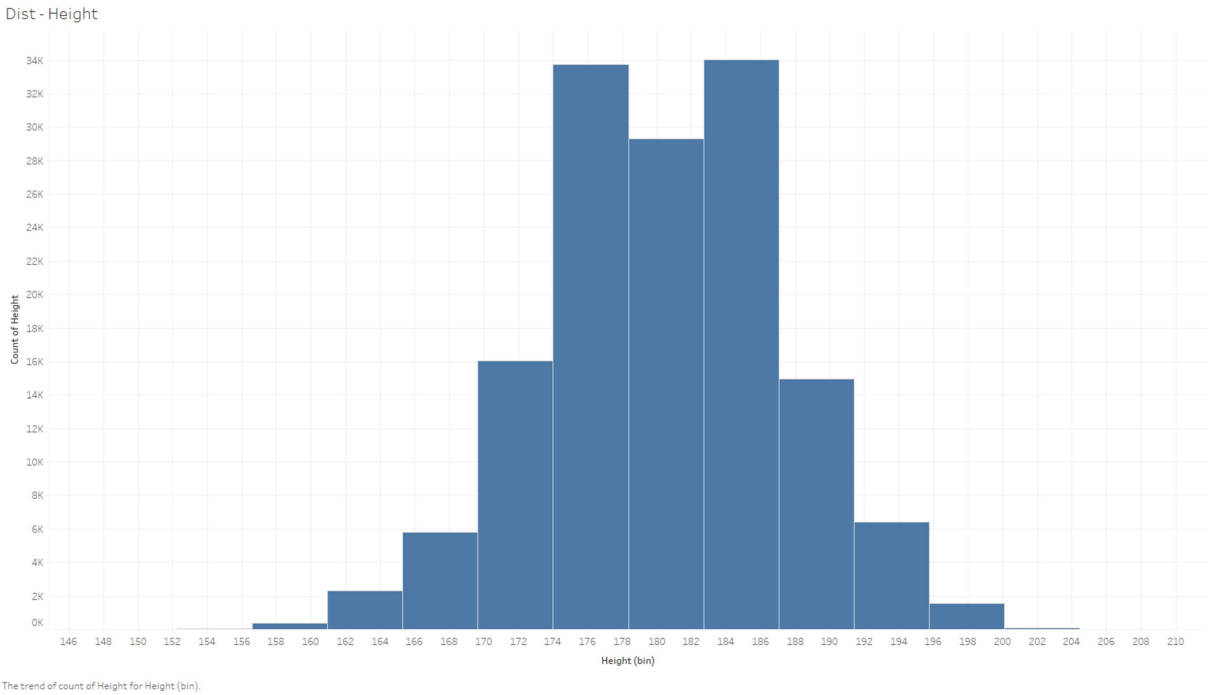
Club. Colour shows details about Club. Size shows sum of CA. The marks are labelled by Club. Details are shown for Division.The view is filtered on DivisionandExclusions (Club,Division).The Division filter keeps Bundesliga 2.The Exclusions (Club,Division) filter keeps 14,756 members.

Another brief interesting look is at the same plot for Ability by nation.

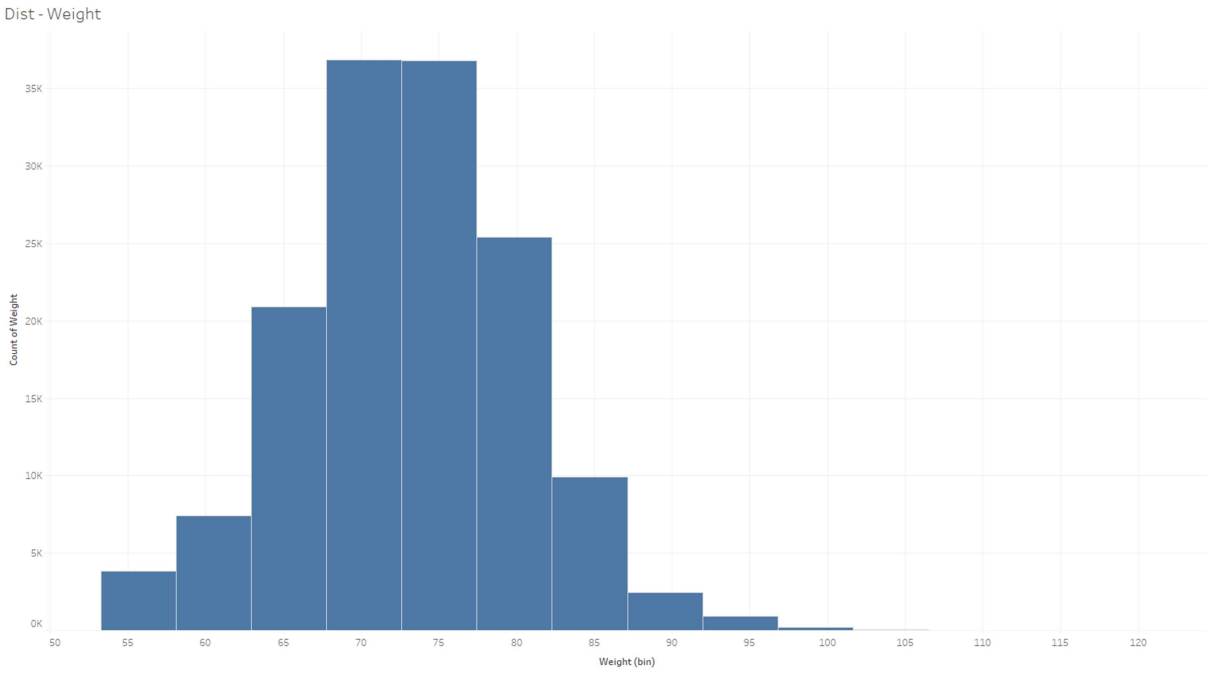


Given that generally speaking Argentina, Brazil, France, Germany, Spain, and Italy are considered to have some fantastic players. The above is a very nice quick visual representation of how those countries fare in terms of total ability.

Finally distributions of Height and Weight were produced as part of the exploration stage. Interestingly to note below, there appears to be less “middle ground” with respect to height. Suggesting that smaller and taller players enter football more generally, as opposed to those of a “typical” height; perhaps this has to do with shorter players being faster and taller players being better in the air. Specialisation in one or the other may be more key to a successful professional career than being somewhere in between.



The trend of count of Height for Height (bin).



The trend of count of Weight for Weight (bin).

Implementation

The Data Set

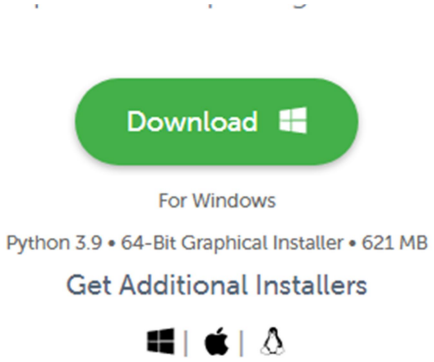
The Data Set can be found at https://www.kaggle.com/datasets/ktyptorio/football-manager-2020 and is readily downloadable as a single file in .csv format for easy use.

Why Jupyter Notebook?

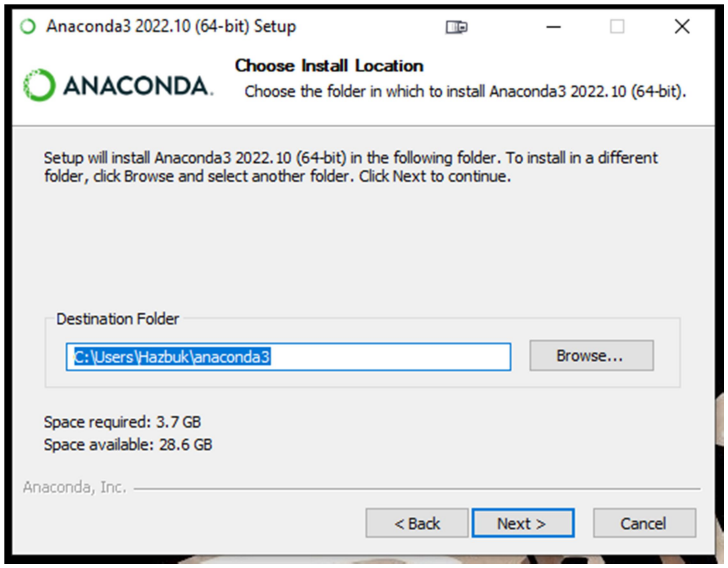
Jupyter Notebook is a powerful Data Science tool generally. It supports multiple languages including R, C++, Ruby and Python all natively. Each block of code is contained within a separate "cell" allowing for easy experimentation without breaking all of your work and when errors do occur it is easy to see exactly what it relates to.

Installing Anaconda and Jupyter Notebook

Anaconda can be downloaded and installed via the following link: https://www.anaconda.com/



Simply click install and follow the instructions on screen

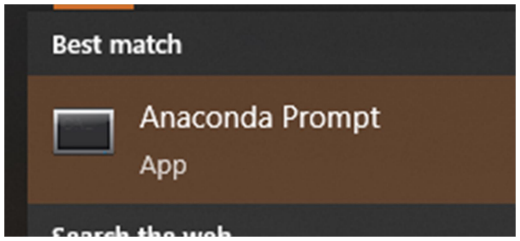


Once the installation is complete you should be ready to move onto the next step of installing PySpark. Anaconda comes with many packages pre-installed specifically for data science. Unfortunately PySpark is not one of them; so some work is needed.

Installing PySpark

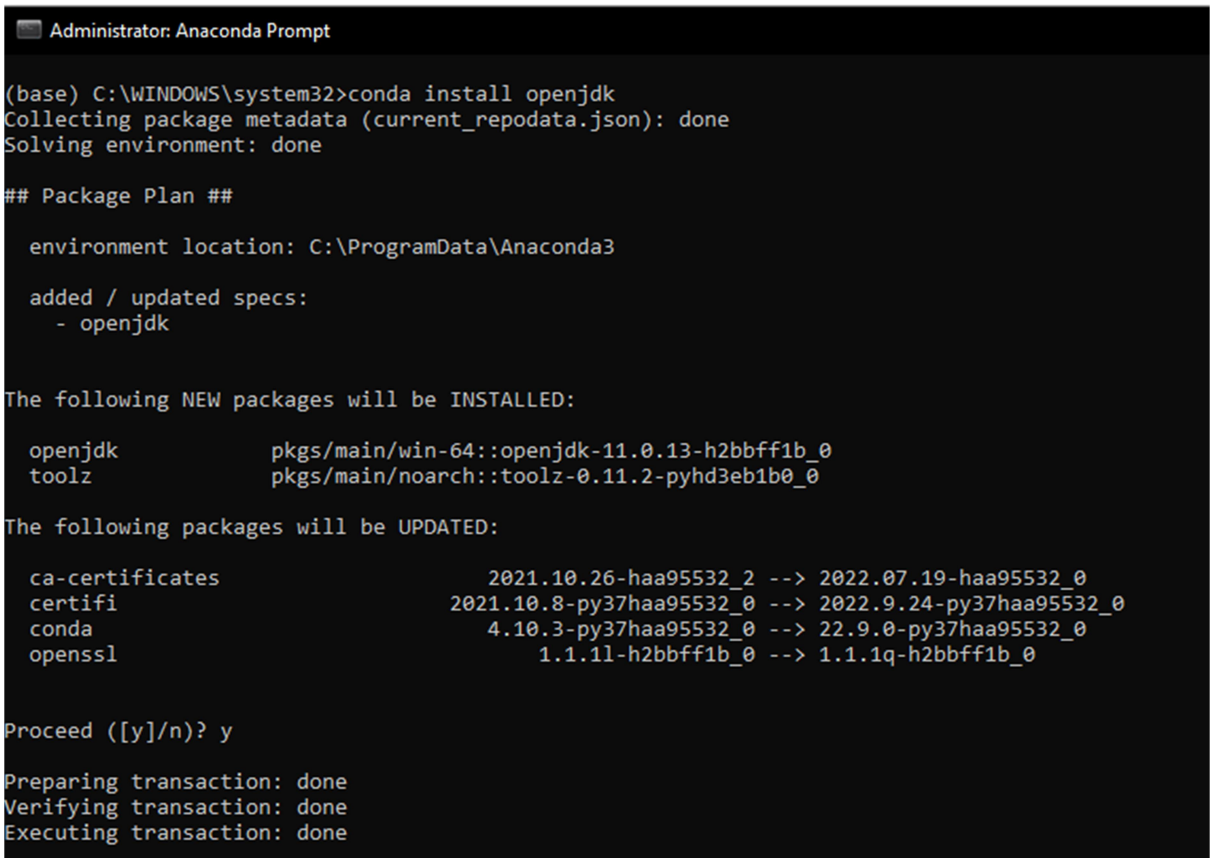
I have been using Jupyter throughout the course thus far and in keeping with this I chose to use it again for this module and find a way to install PySpark within Jupyter Notebooks. I found a very quick, easy and useful guide for installing PySpark and running it through Jupyter and followed it almost exactly (NNK, 2022).

Firstly we open the Anaconda Prompt (this is automatically installed as part of the Anaconda Python Distribution



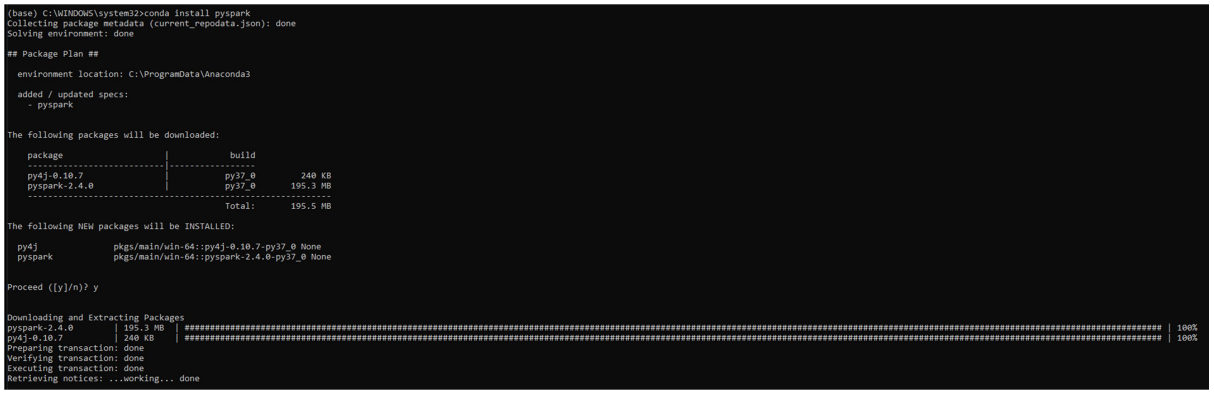
Once opened we install openjdk via the Anaconda Command Prompt using the code below:

```
conda install openjdk
```



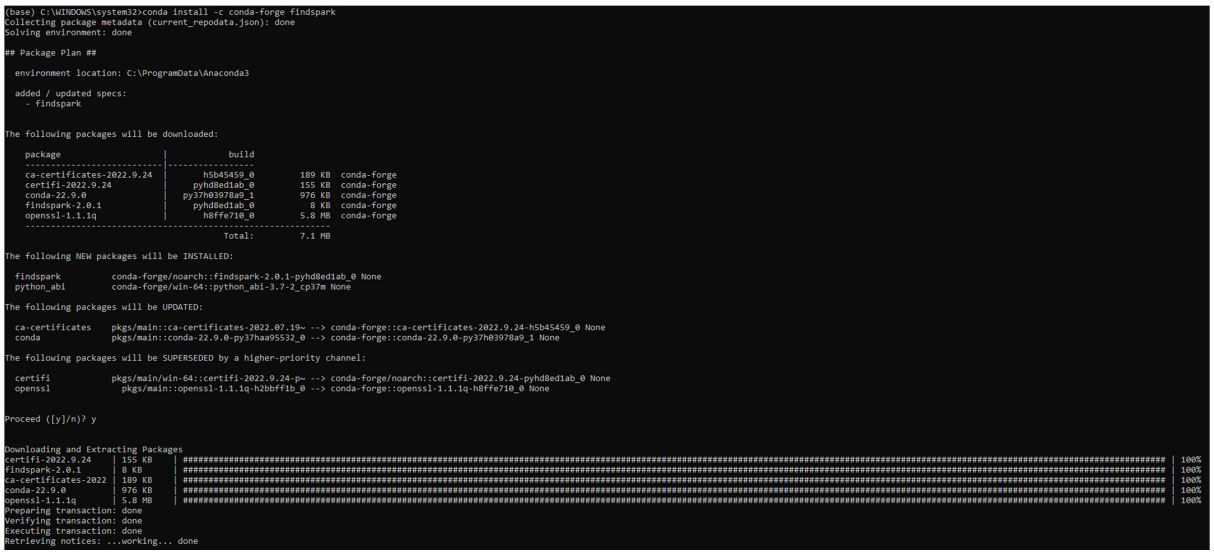
Next we install PySpark via the same command line using the code below:

```
conda install pyspark
```



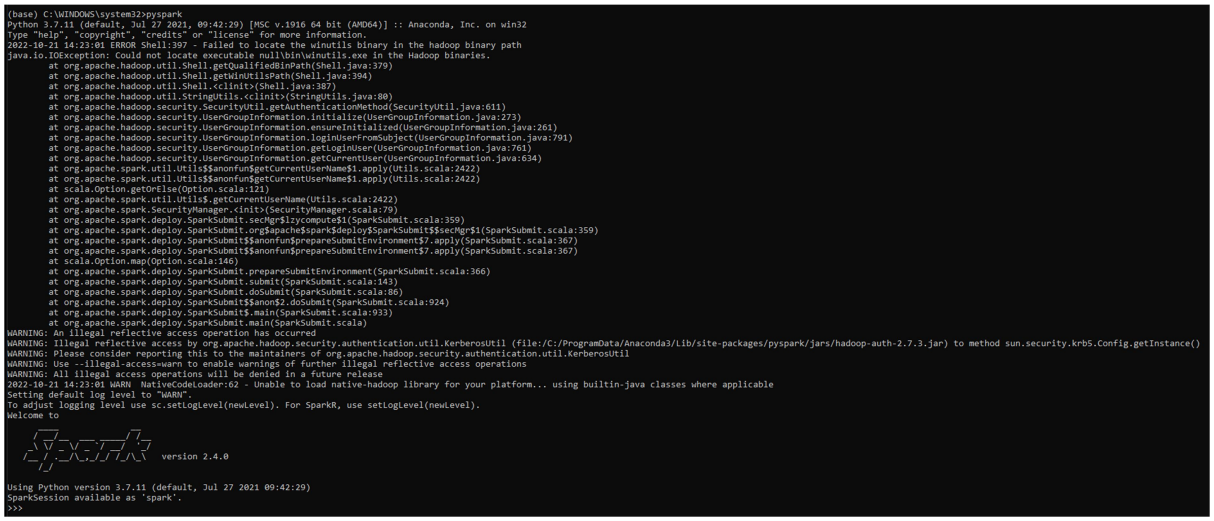
Next we install FindSpark via the same method, using the code below:

```
conda install –c conda-forge findspark
```

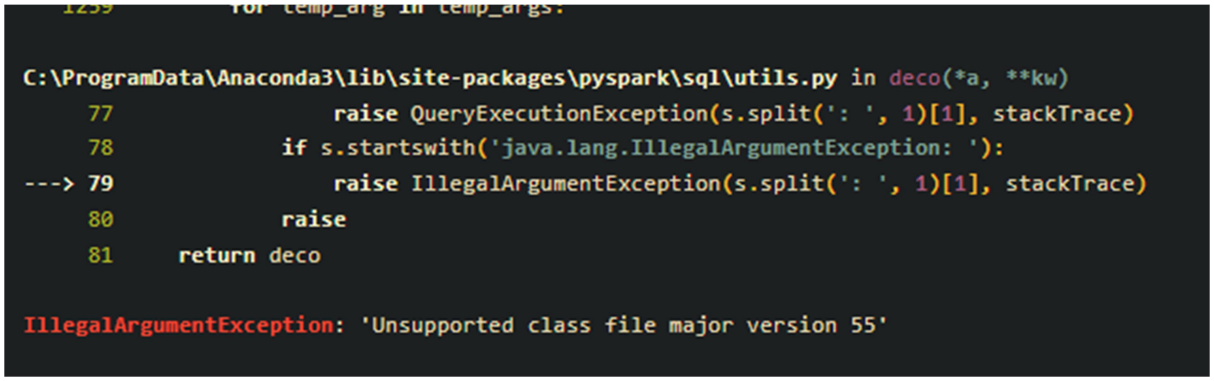


We are now ready to test that PySpark has installed correctly by running the below in the command line:

```
pyspark
```



As part of my individual analysis I encounter the following error:

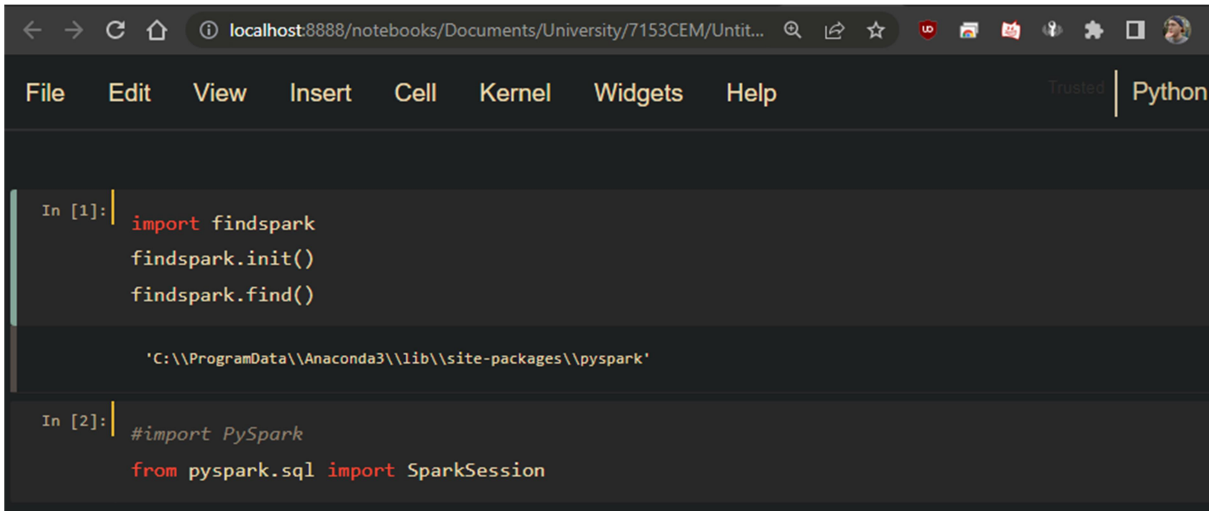


This relates to PySpark not supporting Java versions past version 8, should you encounter the same issue this can be solved as follows. Essentially we are downgrading the java version to 8 and thus, solving the issue.

```
conda install openjdk=8
```

```
(base) C:\WINDOWS\system32>conda install openjdk=8
Collecting package metadata (current_repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Collecting package metadata (repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Solving environment: done

## Package Plan ##

  environment location: C:\ProgramData\Anaconda3

  added / updated specs:
    - openjdk=8


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    openjdk-8.0.152            |       h7382acf_1        48.6 MB
    ------------------------------------------------------------
                                           Total:        48.6 MB

The following NEW packages will be INSTALLED:

  m2-msys2-runtime   pkgs/msys2/win-64::m2-msys2-runtime-2.5.0.17080.65c939c-3 None
  m2-patch           pkgs/msys2/win-64::m2-patch-2.7.5-2 None

The following packages will be UPDATED:

  conda-build                       3.21.5-py37haa95532_0 --> 3.22.0-py37haa95532_0 None

The following packages will be SUPERSEDED by a higher-priority channel:

  ca-certificates    conda-forge::ca-certificates-2022.9.2~ --> pkgs/main::ca-certificates-2022.07.19-haa95532_0 None
  certifi            conda-forge/noarch::certifi-2022.9.24~ --> pkgs/main/win-64::certifi-2022.9.24-py37haa95532_0 None
  conda              conda-forge::conda-22.9.0-py37h03978a~ --> pkgs/main::conda-22.9.0-py37haa95532_0 None
  openssl            conda-forge::openssl-1.1.1q-h8ffe710_0 --> pkgs/main::openssl-1.1.1q-h2bbff1b_0 None

The following packages will be DOWNGRADED:

  openjdk                             11.0.13-h2bbff1b_0 --> 8.0.152-h7382acf_1 None


Proceed ([y]/n)? y


Downloading and Extracting Packages
openjdk-8.0.152      | 48.6 MB   | ############################################################################ | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
Retrieving notices: ...working... done

C:\WINDOWS\system32>set "JAVA_HOME=C:\Program Files\Java\jdk1.8.0_351\jdk1.8.0_351"

C:\WINDOWS\system32>set "JAVA_HOME_CONDA_BACKUP=C:\Program Files\Java\jdk1.8.0_351\jdk1.8.0_351"

C:\WINDOWS\system32>set "JAVA_HOME=C:\ProgramData\Anaconda3\Library"
```

PySpark is now importable as a package within Jupyter Notebook as shown below:



(For clarity that this is my install I have included the header which shows my google login, with my photograph, on the localhost instance of Jupyter – this can be ratified in Appendix A)

Data Pre-Processing

We start by importing the necessary packages for our analysis and data cleaning/wrangling.

In [2]:
```
import pandas as pd
import numpy as np

import pyspark
from pyspark.rdd import RDD
from pyspark.sql import Row
from pyspark.sql import DataFrame
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql import functions
from pyspark.sql.functions import lit, desc, col, size, array_contains\
, isnan, udf, hour, array_min, array_max, countDistinct
from pyspark.sql.types import *

from pyspark.ml  import Pipeline
from pyspark.sql.functions import mean,col,split, col, regexp_extract, when, lit
```

After importing our packages we get our dataset from the local area and read it using PySpark.

In [4]:
```
fm_csv = 'C:/Users/Hazbuk/Documents/University/7153CEM/datafm20.csv'
```

In [5]:
```
fm_data = spark.read.csv(fm_csv,
                         inferSchema=True,
                         header=True)
```

The first step in our cleansing process is to check if any rows are duplicates

```
In [7]:
#count the number rows
Original = fm_data.count()
print("Original Rows: ", Original)
#count the number of potential duplicates (removing later if necessary)
dupe_count = fm_data.dropDuplicates().count()

#Calculate how many duplicate rows exist
Duplicated = Original - dupe_count
print("Duplicated Rows: ", Duplicated)
```

```
Original Rows:  144750
Duplicated Rows:  0
```

```
In [15]:
#count the columns
print("Number of columns: ", len(fm_data.columns))
```

```
Number of columns:  64
```

```
In [17]:
NA = fm_data.dropDuplicates().dropna(
    how="any")# use how="all" for missing data in the entire column
MissingVals = Original - NA.count()
print("Number of N/As: ", MissingVals)
```

```
Number of N/As:  0
```

There were not duplicated rows in the data and so nothing has been removed. We also check that the shape of the DataFrame is as we would expect, i.e. the rows and columns match to our original data set, which they do.

As we saw in our Table analysis, we can also see the vast number of Unknowns within “Club”

In [20]:
```
#The number of players by Club
fm_data.groupBy('Club').count().sort("count",ascending=False).show()
```

```
+----------+-----+
|      Club|count|
+----------+-----+
|   Unknown|20717|
|  Selangor|  137|
|    VÃ0lez|  124|
|     Perak|  122|
|Terengganu|  116|
|     Genoa|  107|
|  Sassuolo|  107|
|    Torino|  106|
|       VIT|  105|
|  Johor DT|  103|
|  Juventus|  103|
|     Inter|  102|
| Sampdoria|  102|
|  EC Bahia|  101|
|    Dinamo|  101|
|  Aldosivi|   95|
|  Sporting|   93|
|      Boca|   93|
|     Milan|   93|
|      Roma|   91|
+----------+-----+
only showing top 20 rows
```

In order to filter these out we use some filtering and print out how many were removed to assure ourselves they have all been deleted.

In [34]:
```
#Remove players with "Unknown" clubs from the data set
fm_data_filtered = fm_data.filter(fm_data['Club']!="Unknown")

unknowns = Original - fm_data_filtered.count()
print("Unknowns removed: ", unknowns)
```

```
Unknowns removed:  20717
```

From Tableau we can additionally note that the “Based” column, which relates to which country a player currently resides in actually contains many effective duplicates.

For example, above, we see Albania has several different categories in relation to its division, however, this is already recorded elsewhere so we would like to remove this. Fortunately, upon inspection, we see that all duplications are of a similar format, namely being that they contain some additional information in brackets after the country in which they are based.

This can be removed using the PySpark code below and regular expression to remove everything after the “(“ and the space before it.

```
In [37]: #Using tableau as part of the pre-processing analysis we see that the "Based" column actually contains
         #several duplications with respect to location (requiring the division for example)
         #we will attempt to clean this up as best as possible using regular expressions

         from pyspark.sql.functions import regexp_replace

         fm_data_filtered = fm_data_filtered.withColumn('Based',regexp_replace('Based','\W\(.+',''))
```

(sankaran, 2020)

Next we look at some other potentially troublesome columns, Position contains a large variety of value, too many to properly categorise. This is because it relates to any position a player *could* properly fulfil the role of. As such, we drop this column and instead use Best Pos and Best Role as better indicators of Position.

```
In [39]: #Position details every possible position a player can play, and contains a large variety of combinations,
         #781, as shown below, this is clearly too many to properly categorise, as such "Best Pos" may be a better
         #column to use. As such we will drop "Position" as it is very messy when compared to "Best Pos"
         print('Combinations of Position:', fm_data_filtered.select('Position').distinct().count())

         fm_data_filtered = fm_data_filtered.drop(*['Position'])

          Combinations of Position: 781

In [40]: #As a check, the below shows the distinct best role and best position categories, these are clearly better
         print('Distinct Best Pos:', fm_data_filtered.select('Best Pos').distinct().count())
         print('Distinct Best Role:', fm_data_filtered.select('Best Role').distinct().count())

          Distinct Best Pos: 14
          Distinct Best Role: 44
```

As previously discussed, Club and Division may be better used as segmentations for separate models as they contain a large number of unique values and the disparity between different divisions and clubs is seemingly quite large. For the purposes of this report these are not included, however, further analysis could be undertaken to properly explore the impact of these elements and produce individual models for each significantly different cluster.

```
In [42]: #Nation and the country a player is Based in would potentially have some impact on the value
         #of a player, as such these will be left in the analysis
         print('Distinct Nations:', fm_data_filtered.select('Nation').distinct().count())
         print('Distinct "Based":', fm_data_filtered.select('Based').distinct().count())
         print('-----')
         #However Club and Division, while having impact, are more an indicator of the fact a division / club
         #is more affluent (and thus can afford said player), rather than a direct indicator of a
         #particular players value. As such, these will be excluded also. Particularly also as they are very
         #large in terms of unique values. One posibility would be to restrict to a particular division for
         #a separate analysis and perhaps build individual models for each division.
         print('Distinct Clubs:', fm_data_filtered.select('Club').distinct().count())
         print('Distinct Divisions:', fm_data_filtered.select('Division').distinct().count())

         fm_data_filtered = fm_data_filtered.drop(*['Club','Division'])

          Distinct Nations: 212
          Distinct "Based": 185
          -----
          Distinct Clubs: 10220
          Distinct Divisions: 1215
```

Now that we have cleansed the data where required we need to perform some encoding on our categorical variables. Machine Learning models require vector inputs and not strings. As such, we need to transform the data to use it.

Firstly I set a new name for the machine learning element specifically

```
In [45]: #Set a new frame for the ML Data Set
         fm_data_ml = fm_data_filtered
```

Regression – Pre-processing

By printing the schema of the data set we can quickly identify which columns will need encoding, a truncated version of this is provided below

```
#Print the Schema to check which columns need encoding
fm_data_filtered.printSchema()
#Based, Nation, Best Pos, Best Role, Preferred Foot
```

```
root
 |-- _c0: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Based: string (nullable = true)
 |-- Nation: string (nullable = true)
 |-- Height: integer (nullable = true)
 |-- Weight: integer (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Preferred Foot: string (nullable = true)
 |-- Best Pos: string (nullable = true)
 |-- Best Role: string (nullable = true)
 |-- Value: integer (nullable = true)
```

Now that they are identified we perform encoding as below, firstly by indexing the values as integers instead of strings.

```
In [46]: #We can see which variables are "string" type and thus may need encoding above.
         #Clearly name will not be encoded as it will not be a predictor in our model

         from pyspark.ml.feature import StringIndexer, OneHotEncoderEstimator, VectorIndexer

         #Use StringIndexer to encode the categorical variables:
         #Based, Nation, Best Pos, Best Role, Preferred Foot
         SI_Based = StringIndexer(inputCol='Based',outputCol='Based_Index')
         SI_Nation = StringIndexer(inputCol='Nation',outputCol='Nation_Index')
         SI_BestPos = StringIndexer(inputCol='Best Pos',outputCol='BestPos_Index')
         SI_BestRole = StringIndexer(inputCol='Best Role',outputCol='BestRole_Index')
         SI_PreferredFoot = StringIndexer(inputCol='Preferred Foot',outputCol='PreferredFoot_Index')

         #Perform the indexing transformations
         fm_data_ml = SI_Based.fit(fm_data_ml).transform(fm_data_ml)
         fm_data_ml = SI_Nation.fit(fm_data_ml).transform(fm_data_ml)
         fm_data_ml = SI_BestPos.fit(fm_data_ml).transform(fm_data_ml)
         fm_data_ml = SI_BestRole.fit(fm_data_ml).transform(fm_data_ml)
         fm_data_ml = SI_PreferredFoot.fit(fm_data_ml).transform(fm_data_ml)
```

Once this is completed we can use One Hot Encoding to better encode the data

```
#Next we use One Hot Encoding
OHE = OneHotEncoderEstimator(inputCols=['Based_Index',
                                        'Nation_Index',
                                        'BestPos_Index',
                                        'BestRole_Index',
                                        'PreferredFoot_Index'],
                             outputCols=['Based_OHE',
                                         'Nation_OHE',
                                         'BestPos_OHE',
                                         'BestRole_OHE',
                                         'PreferredFoot_OHE'])

#Perform the OHE transformation
fm_data_ml = OHE.fit(fm_data_ml).transform(fm_data_ml)
```

Having properly encoded our categorical variables, we now assemble them into a vector for use in our models

```
In [51]: #Finally we assemble the vectors for input into our ML models
         from pyspark.ml.feature import VectorAssembler

         #We will exclude Value and Wage from our input columns, since these are what we would like to predict
         assembler = VectorAssembler(inputCols=[  'Height',
                                                  'Weight',
                                                  'Age',
                                                  'CA',
                                                  'PA',
                                                  'Wor',
                                                  'Vis',
                                                  'Thr',
```

```
                                            'BestPos_OHE',
                                            'PreferredFoot_OHE',
                                            'BestRole_OHE',
                                            'Nation_OHE'],
                                outputCol='features')

#Fill any null values
fm_data_ml = fm_data_ml.fillna(0)

#Perform the transformation
final_data = assembler.transform(fm_data_ml)

# view the transformed vector
final_data.select('features').show()
```

```
+--------------------+
|            features|
+--------------------+
|(512,[0,1,2,3,4,5...|
|(512,[0,1,2,3,4,5...|
|(512,[0,1,2,3,4,5...|
|(512,[0,1,2,3,4,5...|
```

The above represents a truncated screenshot of the output and code, the full code is available in Appendix C.

For our final Regression model, we only require two columns: features and Value. As we will use the newly created feature vector to predict Value of our players.

```
In [52]: final_ml = final_data.select(['features','Value'])
```

Regression – Machine Learning

Fitting a Linear Regression model to all the variables produces an obviously unusable equation for predicting the Value, however, just to show that it can be done it is included below. The best approach to rectifying this I believe is to perform some Principal Component Analysis and dimensionality reduction to best find which variables predict the Value and then use those exclusively. However, I have not included that in this report and opted to perform more classification methods after this point. Partly due to not wanting to reduce the data set when the brief is to work on “big data”.

```
In [53]: from pyspark.ml.regression import LinearRegression
         lr = LinearRegression(featuresCol = 'features', labelCol='Value', maxIter=10, regParam=0.3, elasticNetParam
         lr_model = lr.fit(final_ml)

         # Print the coefficients and intercept for generalized linear regression lr_model
         print("Coefficients: " + str(lr_model.coefficients))
         print("Intercept: " + str(lr_model.intercept))

         # Summarize the model over the training set and print out some metrics
         trainingSummary = lr_model.summary
         print("numIterations: %d" % trainingSummary.totalIterations)
         print("objectiveHistory: %s" % str(trainingSummary.objectiveHistory))
         trainingSummary.residuals.show()
         print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
         print("r2: %f" % trainingSummary.r2)

Coefficients: [10955.340508804866,1605.0793744915725,-20159.370321053313,25882.86913312065,7631.504076242643,-0.0,3648.1626826556735,-160
06.04968668395,7044.9003681136,-4921.843847287381,10135.523742677837,933.6792194541514,14485.4723955621,12981.744908994315,5223.34907407
70685,4712.2775973054195,-10122.961751763274,14903.263383445967,8169.253617469579,54399.91206855639,-4676.5499874616735,6375.88420231438
4,36936.4331411457,-8456.012466898746,-54469.111368731334,2302.165667151223,13078.553785059441,2955.2215453522294,0.0,-6410.734463256241,
5855.012871761621,-12090.252140147877,12337.211343793797,17791.826926299582,11787.018493270174,-12697.849820711983,10987.018347604631,152
38.035734752555,86475.60949746006,1193.5326182593894,-18767.214478070477,11209.650739265631,18535.348630595927,-0.0,3066.90594335547,1154
```

For completeness, Generalised Linear Regression was also implemented, with similar results

```
In [54]: from pyspark.ml.regression import GeneralizedLinearRegression

         glr = GeneralizedLinearRegression(featuresCol = 'features', labelCol='Value', family="gaussian", link="ident

         glr_model = glr.fit(final_ml)

         # Print the coefficients and intercept for generalized linear regression glr_model
         print("Coefficients: " + str(glr_model.coefficients))
         print("Intercept: " + str(glr_model.intercept))

         # Summarize the glr_model over the training set and print out some metrics
         summary = glr_model.summary
         print("Coefficient Standard Errors: " + str(summary.coefficientStandardErrors))
         print("T Values: " + str(summary.tValues))
         print("P Values: " + str(summary.pValues))
         print("Dispersion: " + str(summary.dispersion))
         print("Null Deviance: " + str(summary.nullDeviance))
         print("Residual Degree Of Freedom Null: " + str(summary.residualDegreeOfFreedomNull))
         print("Deviance: " + str(summary.deviance))
         print("Residual Degree Of Freedom: " + str(summary.residualDegreeOfFreedom))
         print("AIC: " + str(summary.aic))
         print("Deviance Residuals: ")
         summary.residuals().show()

Coefficients: [7176.94695139132,-153.56611011707875,-25332.124243816826,77646.12444118629,-2621.7293660953856,-14630.995283845734,-12257.
966645387942,-30145.385096355745,-12909.937026674525,-21777.84853024615,5242.171311829227,-30746.571442179462,-14182.127229147485,18289.
19315413066,-12522.923850594756,1894.3260726563283,-47938.473666131584,15379.432954040549,-16538.41520613714,-7751.669629286961,-23935.31
```

Clustering – Pre-processing

The pre-processing required for clustering is slightly different, with the process detailed below.

We start again by assembling a vector of the variables we intend to use for prediction.

```
In [55]: #As before we assemble a Vector to use for prediction, however, we will attempt to predict some
         #categorical variables from player attributes
         #Can we predict from the data what nationality a player is, what their best position is,
         #or what division they belong to?

         #We will start with Best Pos and exclude this from our input columns
         assembler = VectorAssembler(inputCols=[  'Height',
                                                  'Weight',
                                                  'Age',
                                                  'Wage',
                                                  'Value',
                                                  'CA',
                                                  'PA',
                                                  'Wor',
                                                  'Vis',
                                                  'Thr'
```

```
                                                  'BestRole_OHE',
                                                  'Nation_OHE'],
                                         outputCol='features')


         #Perform the transformation
         final_data_cluster_bestpos = assembler.transform(fm_data_ml)

         # view the transformed vector
         final_data_cluster_bestpos.select('features').show()
```

```
+--------------------+
|            features|
+--------------------+
|(496,[0,1,2,3,4,5...|
|(496,[0,1,2,3,4,5...|
|(496,[0,1,2,3,4,5...|
|(496,[0,1,2,3,4,5...|
|(496,[0,1,2,3,4,5...|
|(496,[0,1,2,3,4,5...|
```

Next we need to index the features as shown below using the VectorIndexor

```
In [56]: from pyspark.ml.evaluation import MulticlassClassificationEvaluator
         from pyspark.ml.classification import DecisionTreeClassifier

In [57]: #Vectorise the features
         vectorindexor = VectorIndexer(inputCol="features",
                                       outputCol="indexedFeatures",
                                       maxCategories=10).fit(final_data_cluster_bestpos)

         dt_data = vectorindexor.transform(final_data_cluster_bestpos)
```

Since we are using a categorical output this time, we need to also encode our response variable, in this case we are looking to predict the players “best pos”.

```
In [58]: #Vectorise the label output
         labelIndexer = StringIndexer(inputCol="Best Pos", outputCol="indexedLabel").fit(dt_data)

         dt_data = labelIndexer.transform(dt_data)
```

Clustering – Machine Learning

Once this is completed we are ready to split our data into testing and training data, to fit the model based on our training data and to then predict the outcomes and see how closely our model fits to the real values.

```
In [59]: #split the data into test & train
         test, train = dt_data.randomSplit([0.8,0.2])

In [60]: #Initialise the Decision Tree Classification model
         dt = DecisionTreeClassifier(labelCol="indexedLabel",
                                     featuresCol="indexedFeatures")


         #fit it to the training data
         dt_model = dt.fit(train)

In [61]: #perform predictions on the data
         predictions = dt_model.transform(test)
```

These predictions can now be viewed via a PySpark select statements to produce the below and directly compare our prediction to the real values.

```
In [62]: #select statement to display some of the predictions next to the actuals
         predictions.select("prediction", "indexedLabel", "features").show(5)
```

```
+----------+------------+--------------------+
|prediction|indexedLabel|            features|
+----------+------------+--------------------+
|       1.0|         5.0|(496,[0,1,2,3,4,5...|
|       1.0|         1.0|(496,[0,1,2,3,4,5...|
|       3.0|         3.0|(496,[0,1,2,3,4,5...|
|       1.0|         8.0|(496,[0,1,2,3,4,5...|
|       1.0|         2.0|(496,[0,1,2,3,4,5...|
+----------+------------+--------------------+
only showing top 5 rows
```

As we can see, it doesn't appear to be incredibly accurate at first glance, but, let us be more scientific and calculate the error using the below.

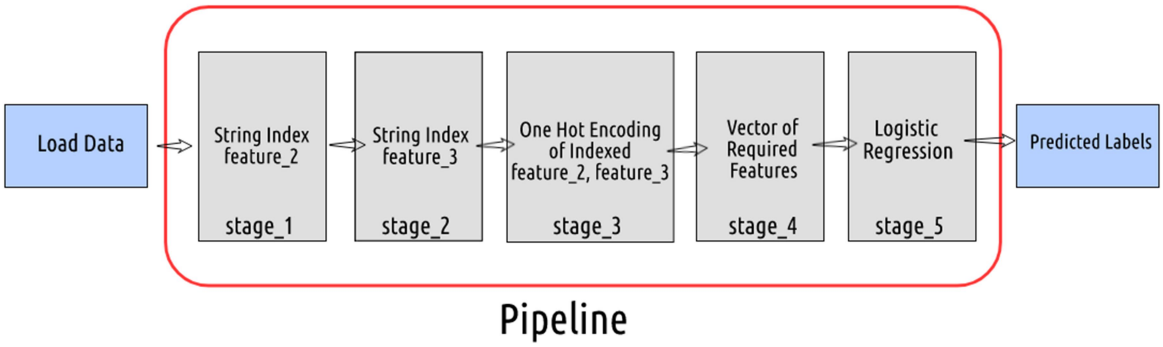
```
In [63]: #Display the error of the model using the Classification Evaluator
         evaluator = MulticlassClassificationEvaluator(
             labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
         accuracy = evaluator.evaluate(predictions)
         print("Test Error = %g " % (1.0 - accuracy))
```

```
Test Error = 0.49321
```

From this we can see we have an error of 49%, as expected, this model is not very accurate.

Pipelines

Pipelines are the true power of PySpark and allow for rapidly reproducible and easy to use testing of many machine learning models. This incorporates all the pre-processing, vectorisation, indexing and the model itself into one single Pipeline.



(Arora, 2019)

The Pipeline set up is very similar to the steps previously shown. We start by setting up the StringIndexers for each of our categorical variables

```
In [65]: #it is also possible to perform the above steps via a pipeline, as below

         #Stages 1-5: Perform indexation on categorical variables that we will use
         stage_1 = StringIndexer(inputCol='Based',outputCol='Based_Index')
         stage_2 = StringIndexer(inputCol='Nation',outputCol='Nation_Index')
         stage_3 = StringIndexer(inputCol='Best Pos',outputCol='BestPos_Index')
         stage_4 = StringIndexer(inputCol='Best Role',outputCol='BestRole_Index')
         stage_5 = StringIndexer(inputCol='Preferred Foot',outputCol='PreferredFoot_Index')
```

Following these stages, we immediately perform our One Hot Encoding on the outputs of these stages

```
#Stage 6: Perform OHE on these indexed variables
stage_6 = OneHotEncoderEstimator(inputCols=[stage_1.getOutputCol(),
                                            stage_2.getOutputCol(),
                                            stage_3.getOutputCol(),
                                            stage_4.getOutputCol(),
                                            stage_5.getOutputCol()],
                                 outputCols= ['Based_encoded',
                                              'Nation_encoded',
                                              'BestPos_encoded',
                                              'BestRole_encoded',
                                              'PreferredFoot_encoded'])
```

Once again, we need to assemble a vector of all our predictors for the model we choose.

```
#Stage 7: Assemble all predictors into a vector for use by the model
stage_7 = VectorAssembler(inputCols=[  'Height',
                                       'Weight',
                                       'Age',
                                       'CA',
                                       'PA',
                                       'BestRole_encoded',
                                       'PreferredFoot_encoded'],
                          outputCol='features')
```

Indexing the vector is performed as follows, keeping the same categories as previously

```
#Stage 8: Index this vector for use in the model with 10 max categories
stage_8 = VectorIndexer(inputCol="features",
                        outputCol="indexedFeatures",
                        maxCategories=10)
```

Index our response variable (label) and then we are ready to implement the model of our choosing as our final stage.

```
#Stage 9: Index the response variable (label)
stage_9 = StringIndexer(inputCol="Best Pos",
                        outputCol="indexedLabel")
```

For this first Pipeline I have chosen to use Decision Tree Classification

```
#Stage 10: Initialise the model of our choosing
stage_10 = DecisionTreeClassifier(featuresCol='indexedFeatures',
                                  labelCol='indexedLabel')
```

Finally once all these stages are prepared, we are ready to assemble them into a pipeline, run our model, fit it to the data and see how our predictions compared to the actuals.

```
#Setup the Pipeline
dt_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline
model = dt_pipeline.fit(fm_data_ml)
#Transform the data
fm_data_ml = model.transform(fm_data_ml)

#View Predicted values vs the Actuals
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)
```

```
+----------+------------+--------------------+
|prediction|indexedLabel|            features|
+----------+------------+--------------------+
|       5.0|         5.0|(507,[0,1,2,3,4,5...|
|       1.0|         1.0|(507,[0,1,2,3,4,5...|
|       1.0|         1.0|(507,[0,1,2,3,4,5...|
|       3.0|         3.0|(507,[0,1,2,3,4,5...|
|       5.0|         8.0|(507,[0,1,2,3,4,5...|
+----------+------------+--------------------+
```

From inspection of the first 5 rows, it looks like we might have a model that has some potential for predicting the Best Position for a player based on almost every attribute in our original data set.

```
In [66]: #Display the error of the model using the Classification Evaluator
         evaluator = MulticlassClassificationEvaluator(
             labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
         accuracy = evaluator.evaluate(fm_data_ml)
         print("Test Error = %g " % (1.0 - accuracy))

          Test Error = 0.301065
```

Check the error in the same manner as previously, we see a 30% error, not perfect but an improvement on the previous.

To show the ease of reproduction I shall include the same method but for Logistic Regression after having produced the above.

```
In [67]: from pyspark.ml.classification import LogisticRegression
         #Using Pipelines we can more easily test several models at once, as below,
         #all other stages remain the same but we change the classification model used and continue

         #Resetting fm_data_ml for pipeline usage
         fm_data_ml = fm_data_filtered
         test, train = fm_data_ml.randomSplit([0.8,0.2])



         #Stage 10: Initialise the model of our choosing
         stage_10 = LogisticRegression(featuresCol='indexedFeatures',labelCol='indexedLabel')

         #Setup the pipeline
         lr_pipeline = Pipeline(stages= [stage_1,
                                         stage_2,
                                         stage_3,
                                         stage_4,
                                         stage_5,
                                         stage_6,
                                         stage_7,
                                         stage_8,
                                         stage_9,
                                         stage_10])

         #Fit the pipeline
         model = lr_pipeline.fit(train)
         #Transform the data
         fm_data_ml = model.transform(train)
```

All that is required is the above, all stages except stage_10 can remain the same, and we simply choose another model to implement over our data.

```
#View the predictions
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)
```

```
+----------+------------+--------------------+
|prediction|indexedLabel|            features|
+----------+------------+--------------------+
|       1.0|         1.0|(459,[0,1,2,3,4,5...|
|       8.0|         8.0|(459,[0,1,2,3,4,5...|
|       1.0|         1.0|(459,[0,1,2,3,4,5...|
|       3.0|         3.0|(459,[0,1,2,3,4,5...|
|       3.0|         3.0|(459,[0,1,2,3,4,5...|
+----------+------------+--------------------+
only showing top 5 rows
```

In [68]:

```
#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g" % (1.0 - accuracy))
```

```
Test Error = 0
```

This time, we find on inspection that our model has performed very well, in fact, too well. With an error of 0% we have very likely ended up over fitting our model in this case and would need to revisit to see what can be done to rectify this issue. Again, most likely the sheer number of variable is making it impossible to predict well and we would need to do some dimensionality reduction.

Findings and Conclusion

Regression

Clearly the Regression analysis was not successful; there are simply too many variables and too much variance in the Values it is attempting to predict. The best course of action with respect to the Regression analysis would be to perform some form of clustering beforehand to produce models for each cluster separately; to remove significant outliers in the data via some sound statistical methods; and to perform some form of dimensionality reduction.

Clustering

I have not included full detail of all iterations of Clustering here, instead, I provide below a table of the error results for each model produced and the code for all models can be found in Appendix C.

Response Variable	Method	Error %
Best Pos	Decision Tree	30%
	Logistic Regression	0%
	Random Forest	12%
Nation	Decision Tree	74%
	Logistic Regression	0%
	Random Forest	31%
Based	Decision Tree	74%
	Logistic Regression	0%
	Random Forest	25%

It is very clear that there is some issue with my Logistic Regression models, these would need to be revisiting to check if implementation is correct or if the data set is appropriate for these models at all without some further wrangling. Random Forest has performed the best, with “Best Pos” being somewhat predictable even with every variable included.

The methods for implementation were straight forward to follow and easily reproducible when testing multiple models and across multiple response variable with very little changes required once initial setup is complete. As such, Pipelines are very clearly powerful within PySpark and would be favoured over individual implementation of each method, unless strictly necessary.

In conclusion the data has shown some interesting trends via our simple inspection from Tableau, some errors have been found which have been rectified in some cases and left in others, and we have managed to produce some models, albeit, not ideal that could be used to produce some fairly inaccurate prediction about a player based on the variables. If I were to revisit the project I would likely spend a great deal more time on truly wrangling the data and segmenting it for better final analysis / results.

References

NNK. (2022, February 19). Install PySpark in Anaconda & Jupyter Notebook. Spark by {Examples}. https://sparkbyexamples.com/pyspark/install-pyspark-in-anaconda-jupyter-notebook/

Data Analysis With Pyspark Dataframe. (n.d.). Www.nbshare.io. Retrieved October 21, 2022, from https://www.nbshare.io/notebook/97969492/Data-Analysis-With-Pyspark-Dataframe/

sankaran. (2020, January 19). Data Wrangling in Pyspark with Regex. Analytics Vidhya. https://medium.com/analytics-vidhya/data-wrangling-in-pyspark-with-regex-ecda9b8f6256

Classification and regression - Spark 2.4.5 Documentation. (n.d.). Spark.apache.org. https://spark.apache.org/docs/latest/ml-classification-regression.html

Arora, L. (2019, November 19). Building Machine Learning Pipelines using Pyspark. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2019/11/build-machine-learning-pipelines-pyspark/

Appendix A – Errors

```
Py4JJavaError: An error occurred while calling o1832.fit.
: java.lang.OutOfMemoryError: Java heap space
        at scala.reflect.ManifestFactory$$anon$12.newArray(Manifest.scala:141)
        at scala.reflect.ManifestFactory$$anon$12.newArray(Manifest.scala:139)
        at scala.Array$.ofDim(Array.scala:218)
        at org.apache.spark.ml.classification.MultiClassSummarizer.histogram(LogisticRegression.scala:1353)
```

Fixed with setting the spark.driver.memory to be higher on open

```
#create session in order to be capable of accessing all Spark API
spark = SparkSession.builder \
    .master('local[*]') \
    .config("spark.driver.memory", "16g") \
    .appName('Hazbuk') \
    .getOrCreate()
```

Appendix B – Full Code

```
import pandas as pd
import numpy as np

import pyspark
from pyspark.rdd import RDD
from pyspark.sql import Row
from pyspark.sql import DataFrame
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql import functions
from pyspark.sql.functions import lit, desc, col, size, array_contains\
, isnan, udf, hour, array_min, array_max, countDistinct
from pyspark.sql.types import *

from pyspark.ml  import Pipeline
from pyspark.sql.functions import mean,col,split, col, regexp_extract, when, lit

#Create session and allocate additional driver memory
spark = SparkSession.builder \
    .master('local[*]') \
    .config("spark.driver.memory", "16g") \
    .appName('Hazbuk') \
    .getOrCreate()


fm_csv = 'C:/Users/Hazbuk/Documents/University/7153CEM/datafm20.csv'

fm_data = spark.read.csv(fm_csv,
                  inferSchema=True,
                  header=True)

#Quick view of the data
fm_data.show(5)


#count the number rows
Original = fm_data.count()
print("Original Rows: ", Original)
#count the number of potential duplicates (removing later if necessary)
dupe_count = fm_data.dropDuplicates().count()

#Calculate how many duplicate rows exist
Duplicated = Original - dupe_count
print("Duplicated Rows: ", Duplicated)


#count the columns
print("Number of columns: ", len(fm_data.columns))


NA = fm_data.dropDuplicates().dropna(
    how="any")# use how="all" for missing data in the entire column
MissingVals = Original - NA.count()
print("Number of N/As: ", MissingVals)
```

```
#The number of players by Nation
fm_data.groupBy('Nation').count().sort("count",ascending=False).show()


#The number of players by Club
fm_data.groupBy('Club').count().sort("count",ascending=False).show()


#Remove players with "Unknown" clubs from the data set
fm_data_filtered = fm_data.filter(fm_data['Club']!="Unknown")

unknowns = Original - fm_data_filtered.count()
print("Unknowns removed: ", unknowns)


#The number of players by Club
fm_data_filtered.groupBy('Club').count().sort("count",ascending=False).show()


#The average value of players by Club
fm_data_filtered.groupBy('Club').mean('Value').sort("avg(Value)",ascending=False).
show()


#Using tableau as part of the pre-processing analysis we see that the "Based"
column actually contains
#several duplications with respect to location (requiring the division for
example)
#we will attempt to clean this up as best as possible using regular expressions

from pyspark.sql.functions import regexp_replace

fm_data_filtered =
fm_data_filtered.withColumn('Based',regexp_replace('Based','\W\(.+',''))



#Checking this below we can see that the column has now been cleansed and will
more accurately
#represent the country in which a player is based
fm_data_filtered.show(5)


#Position details every possible position a player can play, and contains a large
variety of combinations,
#781, as shown below, this is clearly too many to properly categorise, as such
"Best Pos" may be a better
#column to use. As such we will drop "Position" as it is very messy when compared
to "Best Pos"
print('Combinations of Position:',
fm_data_filtered.select('Position').distinct().count())

fm_data_filtered = fm_data_filtered.drop(*['Position'])


#As a check, the below shows the distinct best role and best position categories,
these are clearly better
```

```
print('Distinct Best Pos:', fm_data_filtered.select('Best
Pos').distinct().count())
print('Distinct Best Role:', fm_data_filtered.select('Best
Role').distinct().count())



#Nation and the country a player is Based in would potentially have some impact on
the value
#of a player, as such these will be left in the analysis
print('Distinct Nations:', fm_data_filtered.select('Nation').distinct().count())
print('Distinct "Based":', fm_data_filtered.select('Based').distinct().count())
print('-----')
#However Club and Division, while having impact, are more an indicator of the fact
a division / club
#is more affluent (and thus can afford said player), rather than a direct
indicator of a
#particular players value. As such, these will be excluded also. Particularly also
as they are very
#large in terms of unique values. One posibility would be to restrict to a
particular division for
#a separate analysis and perhaps build individual models for each division.
print('Distinct Clubs:', fm_data_filtered.select('Club').distinct().count())
print('Distinct Divisions:',
fm_data_filtered.select('Division').distinct().count())

fm_data_filtered = fm_data_filtered.drop(*['Club','Division'])


#Now that general cleansing has been completed, we must encode our categorical
data for use in
#Machine Learning models. Simply because these models will only accept numerical
values.

#Print the Schema to check which columns need encoding
fm_data_filtered.printSchema()
#Based, Nation, Best Pos, Best Role, Preferred Foot



#Set a new frame for the ML Data Set
fm_data_ml = fm_data_filtered

#Dropping the Name column as this will need removing for ML analysis
fm_data_ml = fm_data_ml.drop('Name')



#We can see which variables are "string" type and thus may need encoding above.
#Clearly name will not be encoded as it will not be a predictor in our model

from pyspark.ml.feature import StringIndexer, OneHotEncoderEstimator,
VectorIndexer

#Use StringIndexer to encode the categorical variables:
#Based, Nation, Best Pos, Best Role, Preferred Foot
SI_Based = StringIndexer(inputCol='Based',outputCol='Based_Index')
SI_Nation = StringIndexer(inputCol='Nation',outputCol='Nation_Index')
SI_BestPos = StringIndexer(inputCol='Best Pos',outputCol='BestPos_Index')
```

```
SI_BestRole = StringIndexer(inputCol='Best Role',outputCol='BestRole_Index')
SI_PreferredFoot = StringIndexer(inputCol='Preferred
Foot',outputCol='PreferredFoot_Index')

#Perform the indexing transformations
fm_data_ml = SI_Based.fit(fm_data_ml).transform(fm_data_ml)
fm_data_ml = SI_Nation.fit(fm_data_ml).transform(fm_data_ml)
fm_data_ml = SI_BestPos.fit(fm_data_ml).transform(fm_data_ml)
fm_data_ml = SI_BestRole.fit(fm_data_ml).transform(fm_data_ml)
fm_data_ml = SI_PreferredFoot.fit(fm_data_ml).transform(fm_data_ml)


#Show the resulting Indexes next to their original categories
fm_data_ml.select('Based','Based_Index',
                  'Nation','Nation_Index',
                  'Best Pos','BestPos_Index',
                  'Best Role','BestRole_Index',
                  'Preferred Foot','PreferredFoot_Index').show()


#Next we use One Hot Encoding
OHE = OneHotEncoderEstimator(inputCols=['Based_Index',
                                        'Nation_Index',
                                        'BestPos_Index',
                                        'BestRole_Index',
                                        'PreferredFoot_Index'],
                             outputCols=['Based_OHE',
                                         'Nation_OHE',
                                         'BestPos_OHE',
                                         'BestRole_OHE',
                                         'PreferredFoot_OHE'])

#Perform the OHE transformation
fm_data_ml = OHE.fit(fm_data_ml).transform(fm_data_ml)


#Show the resulting Indexes next to their original categories
fm_data_ml.select('Based','Based_OHE',
                  'Nation','Nation_OHE',
                  'Best Pos','BestPos_OHE',
                  'Best Role','BestRole_OHE',
                  'Preferred Foot','PreferredFoot_OHE').show()


#Finally we assemble the vectors for input into our ML models
from pyspark.ml.feature import VectorAssembler

#We will exclude Value and Wage from our input columns, since these are what we
would like to predict
assembler = VectorAssembler(inputCols=[  'Height',
                                         'Weight',
                                         'Age',
                                         'CA',
                                         'PA',
                                         'Wor',
                                         'Vis',
                                         'Thr',
                                         'Tec',
                                         'Tea',
```

```
                                        'Tck',
                                        'Str',
                                        'Sta',
                                        'TRO',
                                        'Ref',
                                        'Pun',
                                        'Pos',
                                        'Pen',
                                        'Pas',
                                        'Pac',
                                        '1v1',
                                        'OtB',
                                        'Nat',
                                        'Mar',
                                        'L Th',
                                        'Lon',
                                        'Ldr',
                                        'Kic',
                                        'Jum',
                                        'Hea',
                                        'Han',
                                        'Fre',
                                        'Fla',
                                        'Fir',
                                        'Fin',
                                        'Ecc',
                                        'Dri',
                                        'Det',
                                        'Dec',
                                        'Cro',
                                        'Cor',
                                        'Cnt',
                                        'Cmp',
                                        'Com',
                                        'Cmd',
                                        'Bra',
                                        'Bal',
                                        'Ant',
                                        'Agi',
                                        'Agg',
                                        'Aer',
                                        'Acc',
                                        'Based_Index',
                                        'Nation_Index',
                                        'BestPos_Index',
                                        'BestRole_Index',
                                        'PreferredFoot_Index',
                                        'Based_OHE',
                                        'BestPos_OHE',
                                        'PreferredFoot_OHE',
                                        'BestRole_OHE',
                                        'Nation_OHE'],
                            outputCol='features')

#Fill any null values
fm_data_ml = fm_data_ml.fillna(0)

#Perform the transformation
final_data = assembler.transform(fm_data_ml)
```

```
# view the transformed vector
final_data.select('features').show()

final_ml = final_data.select(['features','Value'])


from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol = 'features', labelCol='Value', maxIter=10,
regParam=0.3, elasticNetParam=0.8)
lr_model = lr.fit(final_ml)

# Print the coefficients and intercept for generalized linear regression lr_model
print("Coefficients: " + str(lr_model.coefficients))
print("Intercept: " + str(lr_model.intercept))

# Summarize the model over the training set and print out some metrics
trainingSummary = lr_model.summary
print("numIterations: %d" % trainingSummary.totalIterations)
print("objectiveHistory: %s" % str(trainingSummary.objectiveHistory))
trainingSummary.residuals.show()
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)


from pyspark.ml.regression import GeneralizedLinearRegression

glr = GeneralizedLinearRegression(featuresCol = 'features', labelCol='Value',
family="gaussian", link="identity", maxIter=10, regParam=0.3)

glr_model = glr.fit(final_ml)

# Print the coefficients and intercept for generalized linear regression glr_model
print("Coefficients: " + str(glr_model.coefficients))
print("Intercept: " + str(glr_model.intercept))

# Summarize the glr_model over the training set and print out some metrics
summary = glr_model.summary
print("Coefficient Standard Errors: " + str(summary.coefficientStandardErrors))
print("T Values: " + str(summary.tValues))
print("P Values: " + str(summary.pValues))
print("Dispersion: " + str(summary.dispersion))
print("Null Deviance: " + str(summary.nullDeviance))
print("Residual Degree Of Freedom Null: " +
str(summary.residualDegreeOfFreedomNull))
print("Deviance: " + str(summary.deviance))
print("Residual Degree Of Freedom: " + str(summary.residualDegreeOfFreedom))
print("AIC: " + str(summary.aic))
print("Deviance Residuals: ")
summary.residuals().show()


#As before we assemble a Vector to use for prediction, however, we will attempt to
predict some
#categorical variables from player attributes
#Can we predict from the data what nationality a player is, what their best
position is,
#or what division they belong to?
```

```
#We will start with Best Pos and exclude this from our input columns
assembler = VectorAssembler(inputCols=[  'Height',
                                         'Weight',
                                         'Age',
                                         'Wage',
                                         'Value',
                                         'CA',
                                         'PA',
                                         'Wor',
                                         'Vis',
                                         'Thr',
                                         'Tec',
                                         'Tea',
                                         'Tck',
                                         'Str',
                                         'Sta',
                                         'TRO',
                                         'Ref',
                                         'Pun',
                                         'Pos',
                                         'Pen',
                                         'Pas',
                                         'Pac',
                                         '1v1',
                                         'OtB',
                                         'Nat',
                                         'Mar',
                                         'L Th',
                                         'Lon',
                                         'Ldr',
                                         'Kic',
                                         'Jum',
                                         'Hea',
                                         'Han',
                                         'Fre',
                                         'Fla',
                                         'Fir',
                                         'Fin',
                                         'Ecc',
                                         'Dri',
                                         'Det',
                                         'Dec',
                                         'Cro',
                                         'Cor',
                                         'Cnt',
                                         'Cmp',
                                         'Com',
                                         'Cmd',
                                         'Bra',
                                         'Bal',
                                         'Ant',
                                         'Agi',
                                         'Agg',
                                         'Aer',
                                         'Acc',
                                         'Based_OHE',
                                         'PreferredFoot_OHE',
                                         'BestRole_OHE',
                                         'Nation_OHE'],
```

```
                            outputCol='features')

#Perform the transformation
final_data_cluster_bestpos = assembler.transform(fm_data_ml)

# view the transformed vector
final_data_cluster_bestpos.select('features').show()


from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.classification import DecisionTreeClassifier


#Vectorise the features
vectorindexor = VectorIndexer(inputCol="features",
                              outputCol="indexedFeatures",
                              maxCategories=10).fit(final_data_cluster_bestpos)

dt_data = vectorindexor.transform(final_data_cluster_bestpos)

#Vectorise the label output
labelIndexer = StringIndexer(inputCol="Best Pos",
outputCol="indexedLabel").fit(dt_data)

dt_data = labelIndexer.transform(dt_data)


#split the data into test & train
test, train = dt_data.randomSplit([0.8,0.2])


#Initialise the Decision Tree Classification model
dt = DecisionTreeClassifier(labelCol="indexedLabel",
featuresCol="indexedFeatures")

#fit it to the training data
dt_model = dt.fit(train)

#perform predictions on the data
predictions = dt_model.transform(test)

#select statement to display some of the predictions next to the actuals
predictions.select("prediction", "indexedLabel", "features").show(5)

#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g " % (1.0 - accuracy))



#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered

#it is also possible to perform the above steps via a pipeline, as below

#Stages 1-5: Perform indexation on categorical variables that we will use
stage_1 = StringIndexer(inputCol='Based',outputCol='Based_Index')
```

```
stage_2 = StringIndexer(inputCol='Nation',outputCol='Nation_Index')
stage_3 = StringIndexer(inputCol='Best Pos',outputCol='BestPos_Index')
stage_4 = StringIndexer(inputCol='Best Role',outputCol='BestRole_Index')
stage_5 = StringIndexer(inputCol='Preferred Foot',outputCol='PreferredFoot_Index')

#Stage 6: Perform OHE on these indexed variables
stage_6 = OneHotEncoderEstimator(inputCols=[stage_1.getOutputCol(),
                                            stage_2.getOutputCol(),
                                            stage_3.getOutputCol(),
                                            stage_4.getOutputCol(),
                                            stage_5.getOutputCol()],
                                 outputCols= ['Based_encoded',
                                              'Nation_encoded',
                                              'BestPos_encoded',
                                              'BestRole_encoded',
                                              'PreferredFoot_encoded'])

#Stage 7: Assemble all predictors into a vector for use by the model
stage_7 = VectorAssembler(inputCols=[  'Height',
                                       'Weight',
                                       'Age',
                                       'CA',
                                       'PA',
                                       'Wor',
                                       'Vis',
                                       'Thr',
                                       'Tec',
                                       'Tea',
                                       'Tck',
                                       'Str',
                                       'Sta',
                                       'TRO',
                                       'Ref',
                                       'Pun',
                                       'Pos',
                                       'Pen',
                                       'Pas',
                                       'Pac',
                                       '1v1',
                                       'OtB',
                                       'Nat',
                                       'Mar',
                                       'L Th',
                                       'Lon',
                                       'Ldr',
                                       'Kic',
                                       'Jum',
                                       'Hea',
                                       'Han',
                                       'Fre',
                                       'Fla',
                                       'Fir',
                                       'Fin',
                                       'Ecc',
                                       'Dri',
                                       'Det',
                                       'Dec',
                                       'Cro',
                                       'Cor',
```

```
                                        'Cnt',
                                        'Cmp',
                                        'Com',
                                        'Cmd',
                                        'Bra',
                                        'Bal',
                                        'Ant',
                                        'Agi',
                                        'Agg',
                                        'Aer',
                                        'Acc',
                                        'Based_encoded',
                                        'Nation_encoded',
                                        'BestPos_encoded',
                                        'BestRole_encoded',
                                        'PreferredFoot_encoded'],
                            outputCol='features')

#Stage 8: Index this vector for use in the model with 10 max categories
stage_8 = VectorIndexer(inputCol="features",
                        outputCol="indexedFeatures",
                        maxCategories=10)

#Stage 9: Index the response variable (label)
stage_9 = StringIndexer(inputCol="Best Pos",
                        outputCol="indexedLabel")


#Stage 10: Initialise the model of our choosing
stage_10 =
DecisionTreeClassifier(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the Pipeline
dt_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline
model = dt_pipeline.fit(fm_data_ml)
#Transform the data
fm_data_ml = model.transform(fm_data_ml)

#View Predicted values vs the Actuals
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)

#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g " % (1.0 - accuracy))
```

```
from pyspark.ml.classification import LogisticRegression
#Using Pipelines we can more easily test several models at once, as below,
#all other stages remain the same but we change the classification model used and
continue

#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered
test, train = fm_data_ml.randomSplit([0.8,0.2])



#Stage 10: Initialise the model of our choosing
stage_10 =
LogisticRegression(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the pipeline
lr_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline
model = lr_pipeline.fit(train)
#Transform the data
fm_data_ml = model.transform(train)

#View the predictions
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)


#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g" % (1.0 - accuracy))


from pyspark.ml.classification import RandomForestClassifier

#Using Pipelines we can more easily test several models at once, as below,
#all other stages remain the same but we change the classification model used and
continue

#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered
test, train = fm_data_ml.randomSplit([0.8,0.2])

#Stage 10: Initialise the model of our choosing
stage_10 =
RandomForestClassifier(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the pipeline
rf_pipeline = Pipeline(stages= [stage_1,
```

```
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline
model = rf_pipeline.fit(train)
#Transform the data
fm_data_ml = model.transform(train)

#View the predictions
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)


#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g" % (1.0 - accuracy))


#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered
test, train = fm_data_ml.randomSplit([0.8,0.2])

#Predicting Nation using Pipeline with various models
#Stages 1-5: Perform indexation on categorical variables that we will use
stage_1 = StringIndexer(inputCol='Based',outputCol='Based_Index')
stage_2 = StringIndexer(inputCol='Nation',outputCol='Nation_Index')
stage_3 = StringIndexer(inputCol='Best Pos',outputCol='BestPos_Index')
stage_4 = StringIndexer(inputCol='Best Role',outputCol='BestRole_Index')
stage_5 = StringIndexer(inputCol='Preferred Foot',outputCol='PreferredFoot_Index')

#Stage 6: Perform OHE on these indexed variables
stage_6 = OneHotEncoderEstimator(inputCols=[stage_1.getOutputCol(),
                                            stage_2.getOutputCol(),
                                            stage_3.getOutputCol(),
                                            stage_4.getOutputCol(),
                                            stage_5.getOutputCol()],
                                 outputCols= ['Based_encoded',
                                              'Nation_encoded',
                                              'BestPos_encoded',
                                              'BestRole_encoded',
                                              'PreferredFoot_encoded'])

#Stage 7: Assemble all predictors into a vector for use by the model
stage_7 = VectorAssembler(inputCols=[  'Height',
                                       'Weight',
                                       'Age',
                                       'CA',
                                       'PA',
                                       'Wor',
                                       'Vis',
                                       'Thr',
```

```
                                        'Tec',
                                        'Tea',
                                        'Tck',
                                        'Str',
                                        'Sta',
                                        'TRO',
                                        'Ref',
                                        'Pun',
                                        'Pos',
                                        'Pen',
                                        'Pas',
                                        'Pac',
                                        '1v1',
                                        'OtB',
                                        'Nat',
                                        'Mar',
                                        'L Th',
                                        'Lon',
                                        'Ldr',
                                        'Kic',
                                        'Jum',
                                        'Hea',
                                        'Han',
                                        'Fre',
                                        'Fla',
                                        'Fir',
                                        'Fin',
                                        'Ecc',
                                        'Dri',
                                        'Det',
                                        'Dec',
                                        'Cro',
                                        'Cor',
                                        'Cnt',
                                        'Cmp',
                                        'Com',
                                        'Cmd',
                                        'Bra',
                                        'Bal',
                                        'Ant',
                                        'Agi',
                                        'Agg',
                                        'Aer',
                                        'Acc',
                                        'Based_encoded',
                                        'Nation_encoded',
                                        'BestPos_encoded',
                                        'BestRole_encoded',
                                        'PreferredFoot_encoded'],
                            outputCol='features')

#Stage 8: Index this vector for use in the model with 10 max categories
stage_8 = VectorIndexer(inputCol="features",
                        outputCol="indexedFeatures",
                        maxCategories=10)

#Stage 9: Index the response variable (label)
stage_9 = StringIndexer(inputCol="Nation",
                        outputCol="indexedLabel")
```

```
#Stage 10: Initialise the model of our choosing
stage_10 =
DecisionTreeClassifier(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the Pipeline
dt_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline
model = dt_pipeline.fit(fm_data_ml)
#Transform the data
fm_data_ml = model.transform(fm_data_ml)

#View Predicted values vs the Actuals
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)


#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g " % (1.0 - accuracy))



from pyspark.ml.classification import LogisticRegression

#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered
test, train = fm_data_ml.randomSplit([0.8,0.2])



#Stage 10: Initialise the model of our choosing
stage_10 =
LogisticRegression(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the pipeline
lr_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])
```

```
#Fit the pipeline for the training data
model = lr_pipeline.fit(train)
#Transform the data
fm_data_ml = model.transform(train)

#View the predictions
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)


#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g" % (1.0 - accuracy))


from pyspark.ml.classification import RandomForestClassifier

#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered
test, train = fm_data_ml.randomSplit([0.8,0.2])

#Stage 10: Initialise the model of our choosing
stage_10 =
RandomForestClassifier(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the pipeline
rf_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline for the training data
model = rf_pipeline.fit(train)
#Transform the data
fm_data_ml = model.transform(train)

#View the predictions
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)


#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g " % (1.0 - accuracy))



#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered
test, train = fm_data_ml.randomSplit([0.8,0.2])
```

```
#Predicting Based using Pipeline with various models
#Stages 1-5: Perform indexation on categorical variables that we will use
stage_1 = StringIndexer(inputCol='Based',outputCol='Based_Index')
stage_2 = StringIndexer(inputCol='Nation',outputCol='Nation_Index')
stage_3 = StringIndexer(inputCol='Best Pos',outputCol='BestPos_Index')
stage_4 = StringIndexer(inputCol='Best Role',outputCol='BestRole_Index')
stage_5 = StringIndexer(inputCol='Preferred Foot',outputCol='PreferredFoot_Index')

#Stage 6: Perform OHE on these indexed variables
stage_6 = OneHotEncoderEstimator(inputCols=[stage_1.getOutputCol(),
                                            stage_2.getOutputCol(),
                                            stage_3.getOutputCol(),
                                            stage_4.getOutputCol(),
                                            stage_5.getOutputCol()],
                                 outputCols= ['Based_encoded',
                                              'Nation_encoded',
                                              'BestPos_encoded',
                                              'BestRole_encoded',
                                              'PreferredFoot_encoded'])

#Stage 7: Assemble all predictors into a vector for use by the model
stage_7 = VectorAssembler(inputCols=[  'Height',
                                       'Weight',
                                       'Age',
                                       'CA',
                                       'PA',
                                       'Wor',
                                       'Vis',
                                       'Thr',
                                       'Tec',
                                       'Tea',
                                       'Tck',
                                       'Str',
                                       'Sta',
                                       'TRO',
                                       'Ref',
                                       'Pun',
                                       'Pos',
                                       'Pen',
                                       'Pas',
                                       'Pac',
                                       '1v1',
                                       'OtB',
                                       'Nat',
                                       'Mar',
                                       'L Th',
                                       'Lon',
                                       'Ldr',
                                       'Kic',
                                       'Jum',
                                       'Hea',
                                       'Han',
                                       'Fre',
                                       'Fla',
                                       'Fir',
                                       'Fin',
                                       'Ecc',
                                       'Dri',
                                       'Det',
```

```
                                        'Dec',
                                        'Cro',
                                        'Cor',
                                        'Cnt',
                                        'Cmp',
                                        'Com',
                                        'Cmd',
                                        'Bra',
                                        'Bal',
                                        'Ant',
                                        'Agi',
                                        'Agg',
                                        'Aer',
                                        'Acc',
                                        'Based_encoded',
                                        'Nation_encoded',
                                        'BestPos_encoded',
                                        'BestRole_encoded',
                                        'PreferredFoot_encoded'],
                            outputCol='features')

#Stage 8: Index this vector for use in the model with 10 max categories
stage_8 = VectorIndexer(inputCol="features",
                        outputCol="indexedFeatures",
                        maxCategories=10)

#Stage 9: Index the response variable (label)
stage_9 = StringIndexer(inputCol="Based",
                        outputCol="indexedLabel")


#Stage 10: Initialise the model of our choosing
stage_10 =
DecisionTreeClassifier(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the Pipeline
dt_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline
model = dt_pipeline.fit(fm_data_ml)
#Transform the data
fm_data_ml = model.transform(fm_data_ml)

#View Predicted values vs the Actuals
fm_data_ml.select("prediction", "indexedLabel", "features").show()


#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
```

```
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g " % (1.0 - accuracy))


from pyspark.ml.classification import LogisticRegression

#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered
test, train = fm_data_ml.randomSplit([0.8,0.2])



#Stage 10: Initialise the model of our choosing
stage_10 =
LogisticRegression(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the pipeline
lr_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline for the training data
model = lr_pipeline.fit(train)
#Transform the data
fm_data_ml = model.transform(train)

#View the predictions
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)



#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g " % (1.0 - accuracy))



from pyspark.ml.classification import RandomForestClassifier

#Resetting fm_data_ml for pipeline usage
fm_data_ml = fm_data_filtered
test, train = fm_data_ml.randomSplit([0.8,0.2])

#Stage 10: Initialise the model of our choosing
stage_10 =
RandomForestClassifier(featuresCol='indexedFeatures',labelCol='indexedLabel')

#Setup the pipeline
rf_pipeline = Pipeline(stages= [stage_1,
                                stage_2,
```

```
                                stage_3,
                                stage_4,
                                stage_5,
                                stage_6,
                                stage_7,
                                stage_8,
                                stage_9,
                                stage_10])

#Fit the pipeline for the training data
model = rf_pipeline.fit(train)
#Transform the data
fm_data_ml = model.transform(train)

#View the predictions
fm_data_ml.select("prediction", "indexedLabel", "features").show(5)


#Display the error of the model using the Classification Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(fm_data_ml)
print("Test Error = %g " % (1.0 - accuracy))
```