DEAKIN UNIVERSITY

Student Names (Data Scientists)	Muhammad Akbar Husnoo, ID: 219306444 Keerthana Muhunthan, ID: 219499597 Alec Farell, ID: 220003003
Module Name	Modern Data Science
Module Code	SIT742
Assessment	Task 2: Exploration of FIFA 2019 Data.
Lecturer	Associate Professor Dr Gang Li
Report Name	Exploration, Analysis and Prediction of FIFA 2019 Data

TABLE OF CONTENTS

WORK MATRIX

STUDENT NAME	CONTRIBUTIONS			
	PART 1	PART 2	PART 3	REPORT
Muhammad Akbar Husnoo	33.3%	33.3%	33.3%	33.3%
Keerthana Muhunthan	33.3%	33.3%	33.3%	33.3%
Alec Farell	33.3%	33.3%	33.3%	33.3%
TOTAL	100%	100%	100%	100%

Table 1: Work Matrix for each task

1.0. INTRODUCTION

A dataset consisting of attributes of 18,000 FIFA 19 players was released by Kaggle. The data collected included the names of the players, nationalities of the players, age of the players, as well as many others. The assessment, coded utilizing Python 3 and mainly Spark 2.4.0, consists of three parts:

- **Part 1**: Performing exploratory data analysis on the FIFA19 dataset;
- **Part 2**: Performing clustering analysis with the aim of identifying the position profiles of each cluster; &
- **Part 3**: Performing classification analysis with the aim of evaluating the performance of three different algorithms using k-fold cross validation.

This report has been commissioned to address each mandatory section with in-depth technical details as in the following:

2.0. PART 1: EXPLORATORY DATA ANALYSIS

This section deals with exploratory data analysis of the given dataset based on the questions given in the code file as shown in the following:

2.1. 1.1A

2.1.1. Minimum, Maximum and Mean for Age:

During this task, the team was required to compute the minimum, maximum and mean age of the FIFA players.

```python
from pyspark.sql import functions as F

#Descriptive statistics for Age

#Create a function that accepts dataframe values and variable name as parameters for computing max, min and median
def descriptive_summary_measures(dataframe_values, variable_name):
  #try-except block for error handling to prevent abnormal termination of code
  try:
    print('Descriptive Summary Measures for ' + variable_name +':\n')
    #use aggregation for computation
    #computation of min for values
    print ('Minimum Value of ' + variable_name + ':')
    df.agg(F.min(dataframe_values)).show()
    #computation of max for values
    print ('Maximum Value of ' + variable_name + ':')
    df.agg(F.max(dataframe_values)).show()
    #computation of mean for values
    print ('Mean Value of ' + variable_name + ':')
    df.agg(F.mean(dataframe_values)).show()

  except Exception as e:
    logging.error(e)

#Call function that accepts age column and variable name age as argument
descriptive_summary_measures(df['Age'], 'Age')
```

Figure 1: Code Snippet for 1.1A Part 1

From the above code screenshot, a function was created that accepts data frame values and a variable name. The use of effective error-handling (try-except block) prevents abnormal termination of the operation. By calling the function and parsing the required arguments, the descriptive summary measures (minimum, maximum and mean) were computed and displayed as shown in the following.

```
Descriptive Summary Measures for Age:

Minimum Value of Age:
+--------+
|min(Age)|
+--------+
|      16|
+--------+

Maximum Value of Age:
+--------+
|max(Age)|
+--------+
|      45|
+--------+

Mean Value of Age:
+------------------+
|          avg(Age)|
+------------------+
|25.122205745043114|
+------------------+
```

Figure 2: Output for 1.1A Part 1

From the above output, it can be seen that the minimum age of the players was 16 years old, the maximum age of the players was 45 years old, and the mean age of the players was approximately 25.1 years old.

2.1.2. Minimum, Maximum and Mean for Overall:

During this task, the team was required to compute the minimum, maximum and mean overall score of the FIFA players.

```
#Descriptive statistics for Overall

#Function Reuse for computation of Min, Max and Median
#Calling function created above with Overall column and variable name Overall as argument
descriptive_summary_measures(df['Overall'], 'Overall')
```

Figure 3: Code Snippet for 1.1A Part 2

From the above code screenshot, it can be seen that the function created in 1.1A Part 1 was re-utilized and called by parsing the required arguments for computing and displaying the minimum, maximum and mean overall score of the FIFA players as shown below:

```
Descriptive Summary Measures for Overall:

Minimum Value of Overall:
+------------+
|min(Overall)|
+------------+
|          46|
+------------+

Maximum Value of Overall:
+------------+
|max(Overall)|
+------------+
|          94|
+------------+

Mean Value of Overall:
+-----------------+
|     avg(Overall)|
+-----------------+
|66.23869940132916|
+-----------------+
```

Figure 4: Output for 1.1A Part 2

From the above output, it can be seen that the minimum overall score was 46, the maximum overall score was 94, and the mean overall score of the FIFA players was approximately 66.2.

2.1.3. Position having highest Average Overall:

During this task, the team was required to compute the FIFA player position having the highest average overall.

```
#Position having highest Avg Overall (sort Avg Overall by position)

#Create a function that accepts variable names and n as parameters for order by
def order_by (target_variable, order_variable, sort_variable, n):
  #try-except block for error handling to prevent abnormal termination of code
  try:
    #order target_variable by order_variable
    df.groupBy(target_variable).agg(F.mean(order_variable)).sort(sort_variable, ascending = [0]).show(n)
  except Exception as e:
    logging.error(e)

#Call function that accepts Position column, Overall column and n = 1 as argument
print('Position having the highest Average Overall by Position:\n')
order_by('Position', 'Overall', 'avg(Overall)', 1)
```

Figure 5: Code Snippet for 1.1A Part 3

From the above code screenshot, a function was created that accepts three variables and n as parameter. The use of effective error-handling (try-except block) prevents abnormal termination of the operation. The grouping using the mean aggregation was performed and sorted in descending order to compute and display the FIFA player position having the highest overall. The function was called by parsing the required arguments to display the results as shown below:

```
Position having the highest Average Overall by Position:

+--------+-----------------+
|Position|     avg(Overall)|
+--------+-----------------+
|      LF|73.86666666666666|
+--------+-----------------+
only showing top 1 row
```

Figure 6: Code Snippet for 1.1A Part 3

From the above output, it can be concluded that the Left Forward (LF) FIFA player position had the highest average overall score of approximately 73.9.

2.1.4. Top 3 countries having highest Average Overall:

During this task, the team was required to compute the top three countries in FIFA having the highest average overall score.

```
#Top 3 countries with highest Avg Overall

#Function Reuse for computation of top 3 countries with highest Avg Overall
#Calling function created above with Overall, and Nationality Column and n = 3
print('Top 3 Countries with the Highest Average Overall:\n')
order_by('Nationality', 'Overall', 'avg(Overall)', 3)
```

Figure 7: Code Snippet for 1.1A Part 4

From the above code screenshot, it can be seen that the function created in 1.1A Part 3 was re-utilized and called by parsing the required arguments for computing and displaying the values as shown below:

```
Top 3 Countries with the Highest Average Overall:

+--------------------+-----------------+
|         Nationality|     avg(Overall)|
+--------------------+-----------------+
|United Arab Emirates|             77.0|
|Central African Rep.|73.33333333333333|
|              Israel|72.14285714285714|
+--------------------+-----------------+
only showing top 3 rows
```

Figure 8: Output for 1.1A Part 4

From the above output, it can be concluded that United Arab Emirates had the highest average overall score of 77.0, followed by Central African Republic with the average overall score of approximately 73.3 and in the third position, Israel with an average overall score of approximately 72.1.

2.2. 1.1B

2.2.1. Average Potentials on Country by Position:

During this task, the team was required to compute the top ten results for average potentials on country by position sorted in alphabetical order.

```
#Average Potentials on Country by Position with ordering the results on country by alphabet (show top 10)

#try-except block for error handling to prevent abnormal termination of code
try:
  #use aggregation and group by to find required values and sort by Nationality in Alphabetical order
  #then show first 10 results
  print('Average Potentials on Country By Position (Sort Country by alphabetical order): \n')
  df.groupBy('Nationality').pivot('Position').agg(F.mean('Potential')).sort('Nationality').show(10)

except Exception as e:
  logging.error(e)
```

Figure 9: Code Snippet for 1.1B Part 1

From the above code snippet, the use of effective error-handling (try-except block) prevents abnormal termination of the operation. The grouping which involved the use of pivot and mean aggregation was performed and sorted by nationality (in alphabetical order) to display the top ten results as shown below:

```
Average Potentials on Country By Position (Sort Country by alphabetical order):

+-----------------+----+-----------------+-----------------+-----------------+----+-----------------+-----------------+
|      Nationality|null|              CAM|               CB|              CDM|  CF|               CM|               GK|
+-----------------+----+-----------------+-----------------+-----------------+----+-----------------+-----------------+
|      Afghanistan|null|             66.0|             null|             null|null|             71.0|             null|
|          Albania|70.0|            70.75|74.33333333333333|             69.5|null|            71.75|             77.5|
|          Algeria|null|            74.25|             69.5|            70.25|null|77.66666666666667|             67.6|
|          Andorra|null|             null|             64.0|             null|null|             null|             null|
|           Angola|null|             null|             79.0|             null|null|             null|             null|
|Antigua & Barbuda|null|             null|             null|             null|null|             null|             null|
|        Argentina|70.0|74.78260869565217|72.85858585858585|72.76363636363637|79.0|73.08695652173913|71.76288659793815|
|          Armenia|null|             null|             null|             null|null|             null|             null|
|        Australia|null|            71.25| 69.0952380952381|          66.8125|null|67.71428571428571|             67.0|
|          Austria|64.0|73.16666666666667|         70.28125|69.63157894736842|68.0|68.70588235294117| 68.1590909090909|
+-----------------+----+-----------------+-----------------+-----------------+----+-----------------+-----------------+
only showing top 10 rows
```

Figure 10: Output (sample) for 1.1B Part 1

The above output shows the part of the output of average potentials on country by position, sorted by nationality, in alphabetical order. It can also be seen that there are missing null values in the dataset which, though not catered for in this assessment, should have been pre-processed.

2.2.2. Position having the highest Average Potential for Australia:

During this task, the team was required to compute the FIFA player position having the highest Average Potential for Australia.

```
#Position having highest Avg Potential for Australia

#Filter Dataframe using Nationality Column where value is Australia
#Then show Highest Average Potentials (first 10) in Australia
#try-except block for error handling to prevent abnormal termination of code
try:
  #apply filtering
  df_australia = df[df['Nationality'] == 'Australia']
  print('Dataset containing records of Australia only: \n')
  #show filtered dataset
  df_australia.show()
  #group by position, then aggregate on mean to show highest Avg Potential for Australia only
  print('\nPosition having the highest Avg Potential for Australia Only:\n')
  df_australia.groupBy('Position').agg(F.mean('Potential')).sort('avg(Potential)', ascending = [0]).show(1)

except Exception as e:
  logging.error(e)
```

Figure 11: Code Snippet for 1.1B Part 2

From the above code screenshot, the use of effective error-handling (try-except block) prevents abnormal termination of the operation. A filter was applied to save the values from the whole data frame with Australian nationality only. The grouping carried out using mean aggregation was performed and the average potential calculated was then sorted in descending order to show the position having the highest average potential for Australia.

```
Position having the highest Avg Potential for Australia Only:

+--------+--------------+
|Position|avg(Potential)|
+--------+--------------+
|     RDM|          77.0|
+--------+--------------+
only showing top 1 row
```

Figure 12: Output for 1.1B Part 2

From the above output, it can be concluded that the Right Defensive Midfielder (RDM) position for Australia had the highest average potential score of 77.0.

2.3. 1.1C

2.3.1. Plot Age vs Average Potential and Average Overall:

During this task, the team was required to plot the age of FIFA players on the x-axis while the average potential on age and the average overall on age was plotted on the y-axis.

```
#import required libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

#try-except block for error handling to prevent abnormal termination of code
try:
  #finding the average potential by age
  df_avg_potential_by_age = df.groupBy('Age').agg(F.mean('Potential')).sort('Age').toPandas()
  print('The Average Potential by Age:\n\n' + str(df_avg_potential_by_age))
  #finding the average overall by age
  df_avg_overall_by_age = df.groupBy('Age').agg(F.mean('Overall')).sort('Age').toPandas()
  print('\n\nThe Average Overall by Age:\n\n' + str(df_avg_overall_by_age))

except Exception as e:
  logging.error(e)
```

Figure 13: Code Snippet for 1.1C Part 1 (1)

```
#Plot Age on X-axis vs Average Potential and Average Overall

#try-except block for error handling to prevent abnormal termination of code
try:
  #prepare data per axis for avg potential by age
  age_pot = df_avg_potential_by_age["Age"]
  average_potential = df_avg_potential_by_age["avg(Potential)"]
  #prepare data per axis for avg overall by age
  age_ov = df_avg_overall_by_age["Age"]
  average_overall = df_avg_overall_by_age["avg(Overall)"]
  #include plot title
  plt.title('Line Plot of Age against Average Potential and Average Overall')
  #include x-axis label
  plt.xlabel('Player Age (in Years)')
  #include y-axis label
  plt.ylabel('Average Score')
  #plot lines for average potential and average overall
  plt.plot(age_pot, average_potential, label = "Average Potential")
  plt.plot(age_ov, average_overall, label = "Average Overall")
  #include legend
  plt.legend()
  #include grid Lines
  plt.grid(b=True, which='major', linestyle='-')
  #change plot size
  plt.figure(figsize=(10,5))
  # Generate Plot
  plt.show()
except Exception as e:
  logging.error(e)
```

Figure 14: Code Snippet for 1.1C Part 1 (2)

From the above two code screenshots, the use of effective error-handling (try-except block) prevents abnormal termination of the operation. The groupings for average potential by age and average overall by age were performed using mean aggregation and converted to Pandas data frame type for plotting. The data was then plotted on the same graph as shown in the following.

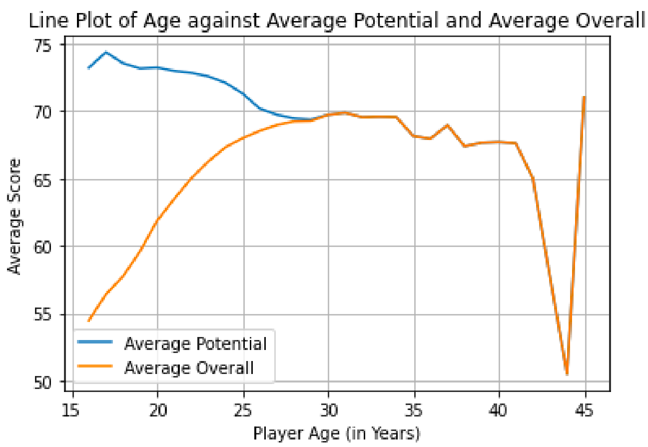

Figure 15: Output plot for 1.1C Part 1

From the above plot output, it can be seen that between the ages of 16 to 30 years old, a unit increase in age tends to increase the average overall while decreasing the average potential of a FIFA player. At approximately the age of 30 years, it is visually evident that both average scores intersect. Post the age of 30 to around 41, both scores of the average FIFA player then gradually decreases in a fairly erratic manner. Between the ages of 41 and 44, there is a sharp drop in both scores which then experience a sudden rise between the ages of 44 and 45, likely due an outlier in older players.

2.3.2. Age when the players fully realise their full potential in general:

During this task, the team was required to compute the exact age whereby, in general, the FIFA players fully realised their potential.

```python
#Find the Intersect (When, on average, players fully realise their potential)
'''
Referenced from https://stackoverflow.com/questions/41247600/subset-pandas-dataframe-by-overlap-with-another
'''
#try-except block for error handling to prevent abnormal termination of code
try:
  #Merges the data frames, and only shows when the Overall and Potential line up.
  Potential_Realised = (df_avg_overall_by_age[df_avg_overall_by_age["avg(Overall)"].isin(df_avg_potential_by_age["avg(Potential)"])])

  #Prints all ages where they are merged
  print(Potential_Realised)
  #Prints the first age they merge
  print("")
  print("Age where players fully released their potential in general\n")
  print(Potential_Realised.iloc[[0]])

except Exception as e:
  logging.error(e)
```

Figure 16: Code Snippet for 1.1C Part 2

Players are considered to fully realise their potential, when their average potential score intersects with their average overall score. From the above code screenshots, the use of effective error-handling (try-except block) prevents abnormal termination of the operation. Both the data frames created in 1.1C Part 1 for the average overall by age and the average potential by age were merged to check which values of age have similar average overall and average potential. The first age where both intersect was displayed as shown below.

```
Age where players fully released their potential in general

    Age  avg(Overall)
15   31     69.850071
```

Figure 17: Output for 1.1C Part 2

From the above output, it can be concluded that, in general, the FIFA players fully realise their potentials at the age of 31 years old with both average potential and average overall scores of approximately 69.85.

3.0. PART 2: UNSUPERVISED LEARNING: K-MEANS

This section deals with data preparation for k-means, model building and predictions with the optimum value of k as subsequently shown:

3.1. 2.1: Data Preparation

During this task, the team was required to prepare the data and find the elbow point for k-means clustering model.

```python
from pyspark.sql.functions import when,col

'''
Code Reference: https://stackoverflow.com/questions/41775281/filtering-a-pyspark-dataframe-using-isin-by-exclusion
'''
#Filtering the Postion = GK (Goal Keepers) from column "Position"

df_filtered = df.filter(~col('Position').isin(['GK']))
df_filtered.show()
```

Figure 18: Code Snippet for 2.1 Data Preparation (1)

```python
'''
Code Reference: https://stackoverflow.com/questions/41775281/filtering-a-pyspark-dataframe-using-isin-by-exclusion
'''


#Assigning position features to 3 groups DEF, FWD and MID.
#Defining the position features under 3 lists/groups
DEF= ["LB","LWB","RB","LCB","RCB","CB","RWB"]
FWD=  ["RF","LF","LW","RS","RW","LS","CF","ST"]
MID=  ["LCM","LM","RDM","CAM","RAM","RCM","CM","CDM","RM","LAM","LDM"]

#Creating a new column "Position_Group", with new Position groups of DEf, FWD and MID respectively.
df1 = df_filtered.withColumn("Position_Group", when(col("Position").isin(DEF), "DEF")
.when(col("Position").isin(FWD), "FWD")
.when(col("Position").isin(MID), "MID"))

#df_kmeans_new (dataframe) has selected attributes along with "ID","Position_Group".
df_kmeans_new = df1.select("ID","Height(CM)","Weight(KG)", "Crossing", "Finishing", "HeadingAccuracy", "ShortPassing",
         "Volleys", "Dribbling", "Curve", "FKAccuracy", "LongPassing", "BallControl", "Acceleration",
         "SprintSpeed", "Agility", "Reactions", "Balance", "ShotPower", "Jumping", "Stamina", "Strength",
         "LongShots", "Aggression", "Interceptions", "Positioning", "Vision", "Penalties", "Composure",
         "Marking", "StandingTackle", "SlidingTackle","Position_Group")


df_kmeans_new.show()
```

Figure 19: Code Snippet for 2.1 Data Preparation (2)

```python
from pyspark.ml.clustering import KMeans
from pyspark.ml.feature import VectorAssembler

#We select the specific features to FEATURED_COL
FEATURES_COL = ['Height(CM)', 'Weight(KG)',
                'Crossing', 'Finishing', 'HeadingAccuracy',
                'ShortPassing', 'Volleys', 'Dribbling', 'Curve',
                'FKAccuracy', 'LongPassing', 'BallControl',
                'Acceleration', 'SprintSpeed', 'Agility',
                'Reactions', 'Balance', 'ShotPower', 'Jumping',
                'Stamina', 'Strength', 'LongShots', 'Aggression',
                'Interceptions', 'Positioning', 'Vision', 'Penalties',
                'Composure', 'Marking', 'StandingTackle', 'SlidingTackle']

#Vector assembler is a transformer which takes all of the columns specified under FEATURES_COL and combines them into a new vector column.
vecAssembler = VectorAssembler(inputCols=FEATURES_COL, outputCol="features")


#We need to store all features as an array of floats, and store this array as a column called "features". Since we do no longer need the original columns we filter them out with a select statement.
#Creating a new vector column "features"
df_kmeans_ = vecAssembler.transform(df_kmeans_new).select('ID','features')
df_kmeans_.show(3)
```

Figure 20: Code Snippet for 2.1 Data Preparation (3)

Based on the above code screenshots, In order to prepare the dataset for building K means clustering for exploring player segmentation, the Goal Keepers position (Position = 'GK') is removed and the position attributes, ID attributes and skill-set attributes (Height(CM), Weight(KG), Crossing, Finishing, HeadingAccuracy, ShortPassing, Volleys, Dribbling, Curve, FKAccuracy, LongPassing, BallControl, Acceleration, SprintSpeed, Agility, Reactions, Balance, ShotPower, Jumping, Stamina, Strength, LongShots, Aggression, Interceptions, Positioning, Vision, Penalties, Composure, Marking, StandingTackle, SlidingTackle) are used. A new column "Position_Group" is created by defining the position features under three major groups: DEF, FWD and MID. Df_kmeans_new comprises all the skill set attributes listed above, and the new "Position_Group". Prior to implementing Kmeans unsupervised classification, the features (excluding "Position_Group") are combined into a new vector column labelled "features".

3.1.1. 2.1: Elbow Plot and Findings:

During this task, the team was required to compute and display the elbow plot with the aim of extracting useful information from the visualisation.

```
#k-means clustering is a method of vector quantization
#Performing k means clustering uses ID and vectorized columns from df_kmeans_
from pyspark.ml.evaluation import ClusteringEvaluator
import numpy as np

'''
Code Reference: https://spark.apache.org/docs/2.2.0/ml-clustering.html#output-columns
'''
cost = np.zeros(21)

for k in range(2,21):

  # Trains a k-means model.
  kmeans = KMeans().setK(k).setFeaturesCol("features").setSeed(1)
  model = kmeans.fit(df_kmeans_)

  #Evaluate clustering by computing Within Set Sum of Squared Errors.
  cost[k] = model.computeCost(df_kmeans_)
  print("Within Set Sum of Squared Errors = " + str(cost[k]))
```

Figure 21: Code Snippet for 2.1 Elbow Plot (1)

Based on the above code screenshot, a for loop is utilised for the range of k between 2 to 20, to iterate through and compute the Within Set Sum of Square Errors as shown below:

```
Within Set Sum of Squared Errors = 67058646.9558278
Within Set Sum of Squared Errors = 53031866.155392915
Within Set Sum of Squared Errors = 48758334.59110524
Within Set Sum of Squared Errors = 44337975.371917725
Within Set Sum of Squared Errors = 42193239.59105987
Within Set Sum of Squared Errors = 40251049.58504104
Within Set Sum of Squared Errors = 38950427.49213952
Within Set Sum of Squared Errors = 37127641.98059367
Within Set Sum of Squared Errors = 36244129.31306243
Within Set Sum of Squared Errors = 35599956.97193515
Within Set Sum of Squared Errors = 34940214.01414501
Within Set Sum of Squared Errors = 34143985.612030484
Within Set Sum of Squared Errors = 33764930.84182901
Within Set Sum of Squared Errors = 33061013.46015743
Within Set Sum of Squared Errors = 32555911.05973738
Within Set Sum of Squared Errors = 32328703.295462158
Within Set Sum of Squared Errors = 32062594.562362324
Within Set Sum of Squared Errors = 31535147.60443827
Within Set Sum of Squared Errors = 31339566.342239458
```

Figure 22: Output for 2.1 Elbow Plot (1)

The elbow method helps identify the optimal number of clusters by fitting the model through a range of K values. The elbow/the bend of the line chart is a good indication that the underlying model fits the best at that point. The elbow curve is plotted using Total within sum of squares verses the number of k values ranging between 2 and 20 as shown:

```
#To optimize k we cluster a fraction of the data for different choices of k and look for an "elbow" in the cost function.
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import seaborn as sbs
from matplotlib.ticker import MaxNLocator

'''
Code Reference: https://runawayhorse001.github.io/LearningApacheSpark/clustering.html
'''

#Plotting the elbow curve
#Total within sum of squares VS the k value
fig, ax = plt.subplots(1,1, figsize =(8,6))
fig.suptitle("Sum of square errors versus K values")
ax.plot(range(2,20),cost[2:20])
ax.set_xlabel('k')
ax.set_ylabel('cost')
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.show()

#From this curve, the bend of the curve, accounting for the highest difference between adjacent totwss, and hence k = 8
```

Figure 23: Code Snippet for 2.1 Elbow Plot (2)

Based on the above code screenshot, the elbow plot is displayed as follows:

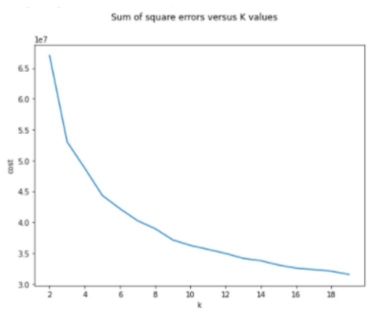

Figure 24: Output for 2.1 Elbow Plot (2)

Based on the above output plot, the elbow of the curve, accounting for the highest difference between adjacent sum of square errors, is at K= 8.

3.2. 2.2: K-means

3.2.1. Findings from clusters:

K-means algorithm assigns data points to a cluster such that the sum of the squared distance between the data points, and the cluster's centroid (arithmetic **means** of all the data points that belong to that cluster), is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster, as we can see below from the cluster center output, representing 8 clusters by arrays of the coordinates of the centroids.

```
Cluster Centers:
[177.3317464   76.31794486  58.41639945  44.02061383  62.79111315
  67.3220339   42.5387082   61.02611086  52.17911131  46.79294549
  63.5510765   65.5758131   65.25240495  66.06596427  64.62757673
  66.20568026  64.10581768  62.28263857  70.52038479  73.82638571
  72.67384333  52.59688502  71.43426477  67.65048099  53.27072836
  57.47457627  48.97938617  64.57627119  66.67155291  68.96472744
  66.88914338]
[187.68026262  81.70680941  39.8684377   30.21252372  68.72675522
  59.71600253  31.19354839  44.54332701  34.72485769  33.42631246
  54.99114485  55.46110057  53.19481341  55.6059456   50.15876028
  62.97975965  50.6116382   51.14231499  68.34977862  64.29095509
  79.52941176  34.24351676  70.56799494  67.06388362  33.600253
  41.73561037  41.05566097  61.12903226  67.29411765  69.6116382
  66.76533839]
[172.46617578  73.21508518  65.18944637  68.70069204  57.79541522
  69.00043253  64.56055363  73.22923875  67.01859862  61.3399654
  60.86245675  72.86548443  76.23096886  75.48442907  76.85856401
  68.82093426  73.3416955   71.57871972  66.4217128   69.08953287
  63.61937716  67.48788927  56.35856401  35.85250865  70.45458478
  67.55363322  65.27032872  68.70804498  36.98140138  34.27465398
  30.68425606]
[179.48039466  76.0082903   36.06083086  24.84124629  55.70029674
  47.25816024  27.14169139  39.07121662  29.75296736  28.67655786
  39.29896142  45.00667656  60.3189911   60.53189911  53.30267062
  53.134273    59.04154303  38.19065282  67.60089021  60.8805638
  67.21216617  25.61053412  57.08234421  56.22922849  31.92062315
  34.70697329  36.51632047  47.67507418  57.19139466  60.84718101
  58.89836795]
[171.83121978  73.63665232  67.45570663  58.56901525  58.09394314
  72.50432633  57.66131026  70.06098063  66.77997528  62.4223321
  69.03461063  72.1223733   69.90688092  69.05644829  71.96744953
  69.5434693   71.48413679  70.77832715  67.38648537  75.37206428
  67.24639473  66.29377833  68.8170581   66.05274001  65.21137206
  68.49938195  59.71693449  69.01771735  63.46147507  66.48413679
  63.407911  ]
[170.91016533  69.059326    55.29234568  55.09580247  43.15901235
  60.04740741  48.40049383  64.16888889  52.72395062  46.47061728
  53.46666667  62.81580247  74.15802469  73.06765432  72.88296296
  54.81975309  74.18222222  58.15407407  57.08493827  59.05530864
  50.88        51.04098765  42.44888889  30.00395062  56.25580247
  56.67012346  52.63111111  55.06469136  35.40345679  33.6108642
  32.67061728]
[179.04405592  77.27376948  41.5253664   64.62852311  62.52367531
  54.67080045  54.12401353  60.1572717   45.93179256  38.34892897
  40.62514092  61.22886133  67.10372041  68.4475761   63.3934611
  58.98816234  61.21364149  63.87373168  66.86583991  62.50281849
  69.78015784  56.90078918  48.6854566   22.34441939  62.61555806
  51.56313416  61.20744081  56.67080045  27.09244645  21.86414882
  19.75479143]
[170.42364272  71.65120522  54.40978964  39.52872168  51.07605178
  60.99029126  37.06432039  58.46197411  45.36407767  40.24959547
  55.51173139  59.90978964  69.55987055  69.1565534   67.27548544
  58.08859223  69.60679612  50.26456311  64.68608414  67.79652104
  60.88754045  42.32686084  59.25444984  56.58576052  50.67435275
  51.5262945   43.85234628  53.67677994  56.21480583  59.43891586
  57.93082524]
Silhouette with squared euclidean distance = 0.22703850068193424
Within Set Sum of Squared Errors = 31339566.342239458
```

Figure 25: Output for 2.2 Part 1

Silhouette approach is a method to predict the squared Euclidean distance and thereby determines the quality of clustering. The best value is 1 and the worst value is -1. The higher the value, the better the quality.

3.2.2. Relationship between Position_Group and a particular Cluster:

The data frame and the counts of each cluster are shown below.

Position_Group	Cluster	count
MID	4	1720
MID	1	117
MID	7	1141
DEF	5	8
MID	0	967
DEF	4	599
DEF	3	1266
MID	3	77
FWD	2	1112
DEF	0	1230
FWD	3	1
FWD	4	71
MID	2	1195
DEF	1	1462
FWD	5	535
DEF	2	4
FWD	6	1648
FWD	7	38
FWD	0	11
MID	5	1486

only showing top 20 rows

Figure 26: Output for 2.2 Part 2 (1)

The same position group can be seen in different clusters. For example, MID players are clustered under cluster 0,1,2,3,4,5 and 7. Similarly DEF players are grouped under 0,1, 2, 3,4 etc. Clusters and the respective player positions are visualized as follows:

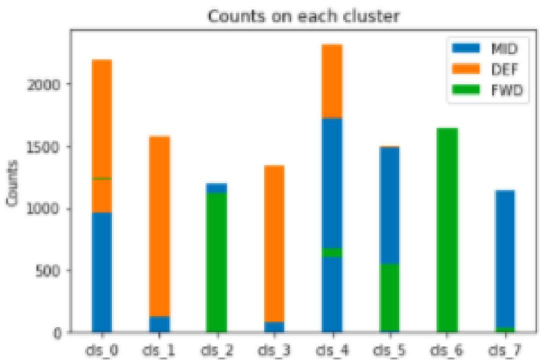

Figure 27: Output for 2.2 Part 2 (2)

Based on the above stacked bar chart output, cluster 6 has only FWD players, and cluster 7 is almost MID players and a minority of FWD. Likewise, Cluster 0, 1 and 3 have a greater number of DEF players. Cluster 4 has the highest number of MID players.

Clustering is done on the basis of the features that have been selected in the beginning, and thereafter predictions are made assigning position groups into different clusters. Clusters are homogenous, therefore when a given cluster has more than one player group, we can conclude the features of those player groups are similar and hence the players are supposed to belong to one cluster. The features of FWD players in cluster 7 are more aligned with the features of

MID players, hence they are grouped into single cluster. Similarly, the cluster 0 and 4, which have lower numbers of FWD players, can be explained as above.

4.0. PART 3: SUPERVISED LEARNING: CLASSIFICATION ON POSITION_GROUP

This section deals with training a logistic regression model and evaluating its scores as well as performing k-fold cross validations on three classification models to identify the best hyper-parameters to be used, as shown in the following:

4.1. 3.2: Training Test Evaluation

4.1.1. Confusion Matrix:

During this task, the team was required to compute the confusion matrix for the logistic regression classifier implemented.

```
#compute Confusion Matrix

'''
Code Reference: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html
'''

#import required confusion_matrix from sklearn library
from sklearn.metrics import confusion_matrix

#try-except block for error handling to prevent abnormal termination of code
try:
  #get the true and predicted position values and convert to list
  true_position = predict_test.select('Target').collect()
  predicted_position = predict_test.select('prediction').collect()
  #compute confusion matrix
  confusion_mat = confusion_matrix(true_position, predicted_position)
  #display confusion matrix
  print( 'The Confusion Matrix for the Logistic Regression Model:\n\n' + str(confusion_mat))

except Exception as e:
  logging.error(e)
```

Figure 28: Code Snippet for 3.2 Part 1

From the above code snippet, it can be seen that the use of effective error-handling (try-except block) prevents abnormal termination of the operation. The true position and the predicted position values were isolated and converted into list format. The lists were parsed as arguments for the confusion matrix computation and displayed as shown below

```
The Confusion Matrix for the Logistic Regression Model:

[[ 821    7  210]
 [   2 1466  307]
 [ 194  240 1637]]
```

Figure 29: Output for 3.2 Part 1

From the above output, it can be concluded that the confusion matrix is as shown in the table with labels:

	FWD	DEF	MID
FWD	821	7	210
DEF	2	1466	307
MID	194	240	1637

Table 2 : Confusion Matrix Table with labels

Based on the confusion matrix, it can be concluded that out of 1038 FWD positions, 821 positions were correctly classified as FWD, 7 wrongly classified as DEF and 210 wrongly classified as MID. Similarly, out of 1775 DEF positions, 1466 positions were correctly classified as DEF, 2 wrongly classified as FWD and 307 wrongly classified as MID. Furthermore, out of 2071 MID positions, 1637 positions were correctly classified as MID, while 194 were wrongly classified as FWD and 240 wrongly classified as DEF.

4.1.2. Classification Report:

During this task, the team was required to compute the precision, recall and F1 scores for the logistic regression classifier implemented.

```
#compute Precision, Recall and F1 score
'''
Code Reference: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html
'''

#import required classification_report from sklearn library
from sklearn.metrics import classification_report

#try-except block for error handling to prevent abnormal termination of code
try:
  #compute classification report
  classification_rep = classification_report(true_position, predicted_position, target_names= ['FWD', 'DEF', 'MID'])
  #display classification report
  print( 'The Classification Report for the Logistic Regression Model:\n\n' + str(classification_rep))
except Exception as e:
  logging.error(e)
```

Figure 30: Code Snippet for 3.2 Part 2

From the above code screenshot, it can be seen that the use of effective error-handling (try-except block) prevents abnormal termination of the operation. A classification report was computed and displayed with the aim of finding the precision, recall and F1 scores for each position category as shown below.

```
The Classification Report for the Logistic Regression Model:

              precision    recall  f1-score   support

         FWD       0.81      0.79      0.80      1038
         DEF       0.86      0.83      0.84      1775
         MID       0.76      0.79      0.77      2071

    accuracy                           0.80      4884
   macro avg       0.81      0.80      0.80      4884
weighted avg       0.80      0.80      0.80      4884
```

Figure 31: Output for 3.2 Part 2

From the above output, the classification report shows the precision, recall and F1 scores for each position category, FWD, DEF and MID. It can be seen that the scores are above 75%, which is considered relatively good. Furthermore, an accuracy of 80% suggests that this model is a relatively suitable model to be used for predicting position groups of players. The overall precision of the model is 80%, the overall F1 score of the model is 80%, and the overall recall score for the model is 80%

4.2. 3.3: K-fold Cross Validation

4.2.1. Code design and running result:

During this task, the team was required to implement 3-fold cross validation for any three classification models of personal preference. Therefore, for this task, the team decided to implement K-fold cross validation for Logistic Regression, Random Forest and Decision Tree classifiers.

```
#import required pyspark libraries
from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
```

Figure 32: Code Snippet for 3.3 Part 1 (1)

```
#K-fold (3-folds) cross validation for Logistic Regression Classifier

'''
Code Reference: https://benalexkeen.com/multiclass-text-classification-with-pyspark/
'''

#initialise multiclass classification evaluator based on accuracy metric
mult_class_evaluator = MulticlassClassificationEvaluator(labelCol= 'Target', predictionCol = 'prediction',
                                                         metricName = 'accuracy')

#define pipeline
pipeline_logistiic_regression = Pipeline(stages= [lr])

#fit train data to pipeline
pipeline_logistiic_regression_model = pipeline_logistiic_regression.fit(train)

#set hyper parameter grid
hyper_parameter_grid_logistic_regression = (ParamGridBuilder()\
                                            .addGrid(lr.regParam, [0.01, 0.1, 0.5])\
                                            .addGrid(lr.maxIter, [10,20,50])\
                                            .addGrid(lr.elasticNetParam, [0.0, 0.5, 0.8])\
                                            .addGrid(lr.aggregationDepth, [2, 5, 10])\
                                            .addGrid(lr.threshold, [0.1, 0.5, 0.9])\
                                            .build())

#try-except block for error handling to prevent abnormal termination of code
try:
  #initialise cross validator with k = 3
  cross_validator_pipeline_logistic_regression = CrossValidator(estimator = pipeline_logistiic_regression,
                                                   estimatorParamMaps = hyper_parameter_grid_logistic_regression,
                                                   evaluator = mult_class_evaluator, numFolds = 3)
  #fit cross validator on training data
  cross_validating_pipeline_logistic_regression_model = cross_validator_pipeline_logistic_regression.fit(train)

  #get the best pipeline  regression model used from the best hyper-parameters used
  best_model_pipeline_logistic_regression = cross_validating_pipeline_logistic_regression_model.bestModel

  #get the logistic regression model from the best pipeline model
  best_model_logistic_regression = best_model_pipeline_logistic_regression.stages[0]

  #find the best hyperparameters used
  print('The best hyper-parameters for tuning this Logistic Regression Model are:\n\n' + str(best_model_logistic_regression.extractParamMap()))

  #use cross-validated best model for predictions
  predictions_using_logistic_regression = best_model_pipeline_logistic_regression.transform(test)

  #evaluate initial model
  print('\n\nAccuracy of initial Logistic Regression Model: ' + str(mult_class_evaluator.evaluate(predict_test)))
  #evaluate cross validated model
  print('\n\nAccuracy of cross validated final Logistic Regression Model: ' + str(mult_class_evaluator.evaluate(predictions_using_logistic_regression)))

except Exception as e:
  logging.error(e)
```

Figure 33: Code Snippet for 3.3 Part 1 (2)

```
#K-fold (3-folds) cross validation for Random Forest Classifier

'''
Code Reference:
1. https://benalexkeen.com/multiclass-text-classification-with-pyspark/
2. https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier
'''

try:
  #define random forest classifier
  random_forest_classifier = RandomForestClassifier(labelCol="Target", featuresCol="Scaled_features")
  #fit training dataset random tree classifier
  initial_random_forest_model = random_forest_classifier.fit(train)
  predict_initial_random_forest = initial_random_forest_model.transform(test)
except Exception as e:
  logging.error(e)

hyper_parameter_grid_random_forest_classifier = (ParamGridBuilder()\
                                                 .addGrid(random_forest_classifier.cacheNodeIds, [True, False])\
                                                 .addGrid(random_forest_classifier.checkpointInterval, [1, 5, 10])\
                                                 .addGrid(random_forest_classifier.impurity, ['gini', 'entropy'])\
                                                 .addGrid(random_forest_classifier.maxDepth, [1, 5, 10])\
                                                 .addGrid(random_forest_classifier.subsamplingRate, [0.1, 0.5, 1.0])\
                                                 .addGrid(random_forest_classifier.minInstancesPerNode, [1, 5, 10])\
                                                 .build())

#try-except block for error handling to prevent abnormal termination of code
try:
  #initialise cross validator with k = 3
  cross_validator_random_forest_classifier = CrossValidator(estimator = random_forest_classifier, estimatorParamMaps = hyper_parameter_grid_random_forest_classifier,
                                                            evaluator = mult_class_evaluator, numFolds = 3)
  #fit cross validator on training data
  cross_validating_random_forest_classifier = cross_validator_random_forest_classifier.fit(train)

  #get the best random forest model used from the best hyper-parameters used
  best_model_random_forest_classifier = cross_validating_random_forest_classifier.bestModel

  #find the best hyperparameters used
  print('The best hyper-parameters for tuning this Random Forest Cross Validated Model are:\n\n' + str(best_model_random_forest_classifier.extractParamMap()))

  #use cross-validated best model for predictions
  predictions_using_random_forest_classifier = best_model_random_forest_classifier.transform(test)

  #evaluate initial model
  print('\n\nAccuracy of initial Random Forest Model: ' + str(mult_class_evaluator.evaluate(predict_initial_random_forest)))
  #evaluate cross validated model
  print('\n\nAccuracy of cross validated final Random Forest Model: ' + str(mult_class_evaluator.evaluate(predictions_using_random_forest_classifier)))

except Exception as e:
  logging.error(e)
```

Figure 34: Code Snippet for 3.3 Part 1 (3)

```
#K-fold (3-folds) cross validation for Decision Tree Classifier

'''
Code References:
1. https://benalexkeen.com/multiclass-text-classification-with-pyspark/
2. https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-trees
'''

try:
  #define random forest classifier
  decision_tree_classifier = DecisionTreeClassifier(labelCol="Target", featuresCol="Scaled_features")
  #fit training dataset random tree classifier
  initial_decision_tree_model = decision_tree_classifier.fit(train)
  predict_initial_decision_tree = initial_decision_tree_model.transform(test)
except Exception as e:
  logging.error(e)

hyper_parameter_grid_decision_tree_classifier = (ParamGridBuilder()\
                                                 .addGrid(decision_tree_classifier.cacheNodeIds, [True, False])\
                                                 .addGrid(decision_tree_classifier.checkpointInterval, [1, 5, 10])\
                                                 .addGrid(decision_tree_classifier.impurity, ['gini', 'entropy'])\
                                                 .addGrid(decision_tree_classifier.maxBins, [20, 32, 40])\
                                                 .addGrid(decision_tree_classifier.minInstancesPerNode, [1, 5, 10])\
                                                 .build())

#try-except block for error handling to prevent abnormal termination of code
try:
  #initialise cross validator with k = 3
  cross_validator_decision_tree_classifier = CrossValidator(estimator = decision_tree_classifier, estimatorParamMaps = hyper_parameter_grid_decision_tree_classifier,
                                                            evaluator = mult_class_evaluator, numFolds = 3)
  #fit cross validator on training data
  cross_validating_decision_tree_classifier = cross_validator_decision_tree_classifier.fit(train)

  #get the best logistic regression model used from the best hyper-parameters used
  best_model_decision_tree_classifier = cross_validating_decision_tree_classifier.bestModel

  #find the best hyperparameters used
  print('The best hyper-parameters for tuning this Decision Tree Cross Validated Model are:\n\n' + str(best_model_decision_tree_classifier.extractParamMap()))

  #use cross-validated best model for predictions
  predictions_using_decision_tree_classifier = best_model_decision_tree_classifier.transform(test)

  #evaluate initial model
  print('\n\nAccuracy of initial Decision Tree Model: ' + str(mult_class_evaluator.evaluate(predict_initial_random_forest)))
  #evaluate cross validated model
  print('\n\nAccuracy of cross validated final Decision Tree Model: ' + str(mult_class_evaluator.evaluate(predictions_using_decision_tree_classifier)))

except Exception as e:
  logging.error(e)
```

Figure 35: Code Snippet for 3.3 Part 1 (4)

From the above code screenshots, it can be seen that the use of effective error-handling (try-except block) prevents abnormal termination of the operation. Whilst a pipeline was utilized and trained for the logistic regression classifier, the other classifiers did not include so. Hyper-parameter grids for each classifier were defined with some values which can be used to tune the model and boost the accuracy of classification. Cross validations were performed using a multi-class classification evaluator and the best models with the best hyper-parameters were computed, and the hyper-parameters were extracted and displayed. The best models for each classifier were then used for predictions and the accuracies of the best models, and related initial models, were computed and displayed as shown.

```
The best hyper-parameters for tuning this Logistic Regression Model are:
{Param(parent='LogisticRegression_8277dac02338', name='aggregationDepth', doc='su

Accuracy of initial Logistic Regression Model: 0.8034398034398035

Accuracy of cross validated final Logistic Regression Model: 0.8662981162981163
```

Figure 36: Output for 3.3 Part 1 (Logistic Regression Output)

From the above output, it can be seen that the best hyper-parameters were displayed as discussed in later sections. The computation of accuracies can therefore conclude that performing cross-validation of a logistic regression classifier boosts the accuracy of predictions by approximately 6%.

```
The best hyper-parameters for tuning this Random Forest Cross Validated Model are:
{Param(parent='RandomForestClassifier_2d68e324a875', name='cacheNodeIds', doc='If false, the algori

Accuracy of initial Random Forest Model: 0.8208435708435708

Accuracy of cross validated final Random Forest Model: 0.8591318591318591
```

Figure 37: Output for 3.3 Part 1 (Random Forest Output)

From the above output, it can be seen that the best hyper-parameters were displayed as discussed in later sections. The computation of accuracies can therefore conclude that performing cross-validation of a Random Forest classifier boosts the accuracy of predictions by approximately 3%.

```
The best hyper-parameters for tuning this Decision Tree Cross Validated Model are:
{Param(parent='DecisionTreeClassifier_fb711054abbb', name='cacheNodeIds', doc='If false, the algorithm wi

Accuracy of initial Decision Tree Model: 0.8208435708435708

Accuracy of cross validated final Decision Tree Model: 0.8091728091728092
```

Figure 38: Output for 3.3 Part 1 (Decision Tree Output)

From the above output, it can be seen that the best hyper-parameters were displayed as discussed in later sections. The computation of accuracies can therefore conclude that performing cross-validation of a Decision Tree classifier diminishes the accuracy of predictions by approximately 2% which is a challenge faced as discussed later.

4.2.2. Hyper-parameter Findings:

The best hyper-parameters extracted from the best Logistic Regression classification model, computed after cross-validation and hyper-parameter tuning, are as tabulated below:

Hyper-parameters	Best Value
regParam	0.01
maxIter	50
elasticNetParam	0.0

aggregationDepth	2
threshold	0.1

Table 3: Best Hyper-parameter values after 3-fold Cross Validation for Logistic Regression

The best hyper-parameters extracted from the best Random Forest classification model, computed after cross-validation and hyper-parameter tuning, are as tabulated below:

Hyper-parameters	**Best Value**
cacheNodeIds	True
checkpointInterval	1
impurity	entropy
maxDepth	10
subsamplingRate	1.0
minInstancesPerNode	1

Table 4: Best Hyper-parameter values after 3-fold Cross Validation for Random Forest

The best hyper-parameters extracted from the best Decision Tree classification model, computed after cross-validation and hyper-parameter tuning, are as tabulated below:

Hyper-parameters	**Best Value**
cacheNodeIds	True
checkpointInterval	1
impurity	gini
maxBins	40
minInstancesPerNode	5

Table 5: Best Hyper-parameter values after 3-fold Cross Validation for Decision Tree

4.2.3. Challenges Faced:

During this stage of supervised learning and implementation of the three aforementioned classification models, some of the challenges faced, which may be subject to low accuracy were:

1. Untidy data in the dataset which can be resolved through robust pre-processing;
2. There may be violations in assumptions of the underlying data which is not catered for;
3. Computational resources allocated by Google Collab (free version), such as RAM, is very limited, which increases the time consumed for performing learning and predictions;
4. The excessive use of TPU as a hardware accelerator sometimes gets the team off Google Collab;

5. Choosing the hyper-parameters values for the creation of the hyper-parameter grids; &
6. Hyper-parameter tuning sometimes results in a decrease in accuracy rate, which is can be seen in the case of the decision tree.

4.2.4. Additional Task on Dataset:

During this task, the team decided to use k-means clustering to cluster and explore the gradings on the potential of FIFA players.

```
#Additional Task 3.3
'''
Code References:
1. https://stackoverflow.com/questions/41775281/filtering-a-pyspark-dataframe-using-isin-by-exclusion
2. https://stackoverflow.com/questions/41775281/filtering-a-pyspark-dataframe-using-isin-by-exclusion
3. https://spark.apache.org/docs/2.2.0/ml-clustering.html#output-columns
'''
#Cluster the potential based on selected features

#Label encoding the column "Potential"
high = list(range(86,101))
middle = list(range (71, 86))
low = list(range(56, 71))
very_low = list(range(45,56))


#Create a new column titled "Potential_Grading", after labelling
df_new = df.withColumn("Potential_Grading", when(col("Potential").isin(high), "high").
                       when(col("Potential").isin(middle), "Middle").
                       when(col("Potential").isin(low), "Low").
                       when(col("Potential").isin(very_low),"Verylow").otherwise("None"))

#Select the features
df_kmeans = df_new.select("ID","Height(CM)","Weight(KG)", "Crossing", "Finishing", "HeadingAccuracy", "ShortPassing",
          "Volleys", "Dribbling", "Curve", "FKAccuracy", "LongPassing", "BallControl", "Acceleration",
          "SprintSpeed", "Agility", "Reactions", "Balance", "ShotPower", "Jumping", "Stamina", "Strength",
          "LongShots", "Aggression", "Interceptions", "Positioning", "Vision", "Penalties", "Composure",
          "Marking", "StandingTackle", "SlidingTackle","Potential_Grading")

#Features to vectorize
FEATURES_COL1 = ["Height(CM)","Weight(KG)", "Crossing", "Finishing", "HeadingAccuracy", "ShortPassing",
          "Volleys", "Dribbling", "Curve", "FKAccuracy", "LongPassing", "BallControl", "Acceleration",
          "SprintSpeed", "Agility", "Reactions", "Balance", "ShotPower", "Jumping", "Stamina", "Strength",
          "LongShots", "Aggression", "Interceptions", "Positioning", "Vision", "Penalties", "Composure",
          "Marking", "StandingTackle", "SlidingTackle"]
```

Figure 39: Code Snippet for 3.3 Additional Task (1)

```
vecAssembler = VectorAssembler(inputCols=FEATURES_COL1, outputCol="features")

#df_kmeans_1 = vecAssembler.transform(df_kmeans).select('ID','features')
df_kmeans_1 = vecAssembler.setHandleInvalid("skip").transform(df_kmeans).select('ID','features')
df_kmeans_1.show(10)
```

Figure 40: Code Snippet for 3.3 Additional Task (2)

```
from pyspark.ml.evaluation import ClusteringEvaluator
import numpy as np

cost = np.zeros(21)

for k in range(2,21):

  # Trains a k-means model.
  kmeans = KMeans().setK(k).setFeaturesCol("features").setSeed(1)
  model1 = kmeans.fit(df_kmeans_1)

  #Evaluate clustering by computing Within Set Sum of Squared Errors.
  cost[k] = model1.computeCost(df_kmeans_1)
  print("Within Set Sum of Squared Errors = " + str(cost[k]))
```

Figure 41: Code Snippet for 3.3 Additional Task (3)

```
fig, ax = plt.subplots(1,1, figsize =(8,6))
fig.suptitle("Sum of square errors versus K values")
ax.plot(range(2,20),cost[2:20])
ax.set_xlabel('k')
ax.set_ylabel('cost')
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.show()
```

Figure 42: Code Snippet for 3.3 Additional Task (4)

```
k = 8

# Trains a k-means model with the optimized k number
kmeans1 = KMeans().setK(8).setSeed(1).setFeaturesCol("features")
model1 = kmeans1.fit(df_kmeans_1)
centers = model1.clusterCenters()

#Once the training has converged we can print out the centroids of the 8 clusters.
print("Cluster Centers: ")
for center in centers:
    print(center)


#Assigning the individual rows to the nearest cluster centroid using transform method,
#which adds 'prediction' column to the dataframe. The prediction value is an integer between 0 and k.
transformed_data = model1.transform(df_kmeans_1)


#selecting only the specific columns from transformed_data
transformed = transformed_data.select('ID', 'Prediction')
rows = transformed.collect()
print(rows[:3])
```

Figure 43: Code Snippet for 3.3 Additional Task (5)

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(spark)

df_pred = sqlContext.createDataFrame(rows)
df_pred.show()


#Joining the two dataframes df_pred and df_kmeans_new on the 'ID' column
df_kmeans_pred_1 = df_pred.join(df_kmeans, 'ID')


#Changing the column name "Prediction" to "Cluster"
df_kmeans_pred_1 = df_kmeans_pred_.withColumnRenamed('Prediction', 'Clusters')

df_kmeans_pred_1.show(10)
```

Figure 44: Code Snippet for 3.3 Additional Task (6)

```
count_on_clusters1 = df_kmeans_pred_1.groupBy(['Potential_Grading','Clusters']).count()
count_on_clusters1.show()
```

Figure 45: Code Snippet for 3.3 Additional Task (7)

```
#Visualise the count on different position groups
#high, middle, low, very_low.

cl_0 = (0,0,913,0)
cl_1 = (0,1352, 1832,0)
cl_2 = (98,1683,0,0)
cl_3 = (25,843, 0,13)
cl_4 = (0,1024,613,1)
cl_5 = (6,948, 1147,8)
cl_6 = (2, 630, 1198,0)
cl_7 = (0,769, 0,4)

#Defining the cluster cordinates
High = np.array([0,0,98,25,0,6,2,0])
Middle = np.array([0,1352,1683,843,1024,948,630,769])
Low = np.array([913,1832,0,0,613,1147,1198,0])
Very_Low = np.array([0,0,0,13,1,8,0,4])
ind = np.arange(len(High))  # the x locations for the groups
width = 0.35       # the width of the bars: can also be len(x) sequence

#Defining the bars
p1 = plt.bar(ind, High, width)
p2 = plt.bar(ind, Middle, width, bottom=High)
p3 = plt.bar(ind, Low, width, bottom=Middle)
p4 = plt.bar(ind, Very_Low, width, bottom=Low)


plt.ylabel('Counts')
plt.title('Counts on each cluster')
plt.xticks(ind, ('cls_0', 'cls_1', 'cls_2', 'cls_3', 'cls_4', 'cls_5', 'cls_6', 'cls_7'))
#plt.yticks(np.arange(0, 5000))
plt.legend((p1[0], p2[0], p3[0],p4[0]), ('High','Middle','Low', 'Very_Low'))
plt.show()
```

Figure 46: Code Snippet for 3.3 Additional Task (8)

Based on the above eight code screenshots, the data was prepared, and the numerical potential values are converted to string values encoded as high, medium, low and very low. A new column entitled ‘Potential_Grading’ was created in which the encoded values of positions were stored. Vectorisation of the selected features which may be influencing a player’s potential is performed, and the Within Set Sum of Squared Errors were computed for each k value, using a for loop. The results were plotted as shown below:

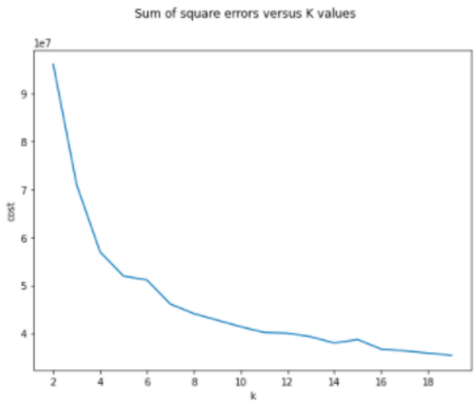

Figure 47: Output for 3.3 Additional Task (1)

Based on the above plot, it can be concluded that the optimum k value, from the graph's elbow, is 8. This optimum k value was used to predict the centroids. Hence, the data was fitted with the potential gradings to determine the count on each cluster with respect to the grades of the potential. A group-by function was used to show the counts on each cluster with respect to the potential grading as shown below:

Potential_Grading	Clusters	count
high	6	2
Middle	2	1683
Low	5	1147
Verylow	3	13
Verylow	5	8
Middle	1	1352
Middle	7	769
Middle	4	1024
Low	4	613
Verylow	4	1
Low	1	1832
high	5	6
Middle	5	948
high	2	98
Low	6	1198
Low	0	913
high	3	25
Middle	6	630
Middle	3	843
Verylow	7	4

Figure 48: Output for 3.3 Additional Task (2)

The above output shows the counts on each cluster with respect to the potential grading and the values obtained were then plotted using a stacked bar chart as shown below:

Figure 49: Output for 3.3 Additional Task (3)

From the above bar chart output, it can be observed that some clusters are predominant, with certain grades of potentials. Cluster 7 and cluster 3 are predominant with Middle potential grade players, cluster 0 with Low potential grade players, and cluster 2 with both Middle potential grade and High potential grade players. Therefore, it can be assumed that the number of medium players in cluster 2 have great potential to evolve into highly skilled players. Cluster 3 has players with the least potential and medium potential.

5.0. CONCLUSION, FINDINGS AND REFLECTION

Throughout this assessment, the team was familiarised with a hands-on experience of exploratory data analysis, clustering using unsupervised k-means, supervised learning and predictions using Logistic Regression Classifier, Random Forest Classifier and Decision Tree Classifier, as well as performing k-fold cross validation to extract the best hyper-parameters. After the analysis, some of the findings revealed were:

- On average, FIFA players will fully realise their potentials at around 31 years of age;
- The position globally with highest average overall score is Left Forward (LF);
- The position with the highest average potential for Australia is Right Defensive Midfielder (RDM);
- United Arab Emirates, Central African Republic and Israel, are the top 3 countries having the highest average overall score;
- From the elbow plot, it was found that the optimal number of clusters (k) for k-means is 8;
- From the bar chart, after fitting, clusters 0 and 4 have predominant numbers of MID and DEF. Since there are two position groups in those clusters, the underlying features are similar for those position groups;
- Clusters 6 and 7 are homogenous (have no mix of positions);
- Midfielders appear to share many elements with Forward (FWD) and Defence (DEF) players, however, defence and forward players do not share any significant similarities between each other;
- The results of the initial predictions for player positions based on relevant player skillsets, without k-fold cross validation and hyper-parameter tuning for all three classifiers implemented, yielded relatively high accuracies of 80% and above in each case;
- Further k-fold validation and hyper-parameter tuning revealed an increase in the accuracies of prediction using the classifiers implemented;
- In comparison with three distinct classifiers with cross validations, Logistic Regression Classifier is relatively the best classifier model for prediction of position groups of FIFA players based on relevant skillsets reaching an accuracy rate of approximately 86.6%; &

- Players with high and low potentials share some similarities with each other in cluster 1, 4, 5 and 6. Players with medium potentials are predominant in cluster 7. Players with high levels of potential share some similarities with players of medium potential, represented by cluster 2.

Reflecting on the usefulness of clustering, we can see that k-means clustering provides an excellent means of clustering player types into groups of shared skillsets. Using machine learning, we can also predict, with great accuracy, what positions a player is likely to play based on their skillsets. Using a dataset of this size, we could theoretically run hundreds of different analyses using the tools we have.