CHESS COURSERA ADVANCED CAPSTONE PROJECT

How to predict who is going to win a chess match

Guglielmo Sanchini

The goal of this project is to predict who is going to win a chess match, in order to provide an online chess platform with useful information to use to make the platform more entertaing and matches more enjoyable

Data Source: [https://www.kaggle.com/datasnaek/chess (https://www.kaggle.com/datasnaek/chess)](https://www.kaggle.com/datasnaek/chess)

Loading

Importing all necessary packages and setting up Spark environment

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter
import joypy
import scipy
```

In [2]:
```
from pyspark.ml.classification import DecisionTreeClassifier, RandomForestClassifi
er, LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split

from pyspark.ml.feature import OneHotEncoderEstimator, StringIndexer, StandardScal
er, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
```

In [4]:
```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras import optimizers, regularizers
from keras.optimizers import Adam
```

In [5]:
```
import findspark
import os

spark_location='/usr/local/Cellar/apache-spark/2.4.3/libexec/' # Set your own
java8_location= '/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Hom
e' # Set your own
os.environ['JAVA_HOME'] = java8_location
findspark.init(spark_home=spark_location)
```

In [48]:
```
try:
    spark.stop()
    print("Spark stopped")
except:
    pass
```

```
Spark stopped
```

In [7]:
```
from pyspark.sql import Row,SQLContext, SparkSession
import pyspark.sql.functions as f
from pyspark.sql.functions import *
from pyspark.sql.types import *

import pyspark
import random

sc = pyspark.SparkContext(appName="SPARK-CHESS")
spark = SparkSession.builder.appName("CHESS").getOrCreate()
```

Loading the dataset

In [8]:
```
df = spark.read.csv('games.csv', header=True)
```

In [9]:
```
sc
```
Out[9]:

SparkContext

[Spark UI (http://10.1.52.19:4040)](http://10.1.52.19:4040)

Version
`v2.4.3`

Master
`local[*]`

AppName
`SPARK-CHESS`

ETL and Feature Creation

Data Exploration and Cleansing

How many rows are in the dataset?

In [10]:
```
df.count()
```
Out[10]:
```
20058
```

How many duplicate rows are in the dataset?

In [11]:
```
df = df.dropDuplicates()
df.count()
```
Out[11]:
```
19629
```

Looking for NaN values

In [12]:
```
df.na.drop().count() #so there are no NaN values
```
Out[12]:
```
19629
```

Column names

In [13]:
```
df.columns
```

Out[13]:
```
['id',
 'rated',
 'created_at',
 'last_move_at',
 'turns',
 'victory_status',
 'winner',
 'increment_code',
 'white_id',
 'white_rating',
 'black_id',
 'black_rating',
 'moves',
 'opening_eco',
 'opening_name',
 'opening_ply']
```

Explanations for some columns

increment code: https://chess.stackexchange.com/questions/18069/what-is-the-increment-in-chess (https://chess.stackexchange.com/questions/18069/what-is-the-increment-in-chess)

how opening_name and opening_eco are related: https://www.365chess.com/eco.php (https://www.365chess.com/eco.php)

Column types

In [14]:
```
df.dtypes
```

Out[14]:
```
[('id', 'string'),
 ('rated', 'string'),
 ('created_at', 'string'),
 ('last_move_at', 'string'),
 ('turns', 'string'),
 ('victory_status', 'string'),
 ('winner', 'string'),
 ('increment_code', 'string'),
 ('white_id', 'string'),
 ('white_rating', 'string'),
 ('black_id', 'string'),
 ('black_rating', 'string'),
 ('moves', 'string'),
 ('opening_eco', 'string'),
 ('opening_name', 'string'),
 ('opening_ply', 'string')]
```

Let's drop some columns which are not relevant for our analysis

In [15]:
```
df = df.drop("created_at", "last_move_at", "white_id", "black_id")
```

In the following lines we separate the increment code column to display in two different columns the fixed time and the additional seconds given at each turn

In [16]:
```
#defining custom functions to split the increment_code column
def funz1(value):
    return value.split("+")[0]

def funz2(value):
    return value.split("+")[1]

udf_funz1 = udf(funz1, StringType())
udf_funz2 = udf(funz2, StringType())

#creating the two new column
df = df.withColumn("increment1", udf_funz1("increment_code"))
df = df.withColumn("increment2", udf_funz2("increment_code"))

#dropping the old columns
df = df.drop("increment_code")
```

We convert all wrong types into the right types, mainly from string to int

In [17]:
```
df = df.withColumn("turns",df["turns"].cast("int"))
df = df.withColumn("white_rating",df["white_rating"].cast("int"))
df = df.withColumn("black_rating",df["black_rating"].cast("int"))
df = df.withColumn("opening_ply",df["opening_ply"].cast("int"))
df = df.withColumn("increment1",df["increment1"].cast("int"))
df = df.withColumn("increment2",df["increment2"].cast("int"))
```

We inspect the new types, to check if they are right

In [18]:
```
df.dtypes
```

Out[18]:
```
[('id', 'string'),
 ('rated', 'string'),
 ('turns', 'int'),
 ('victory_status', 'string'),
 ('winner', 'string'),
 ('white_rating', 'int'),
 ('black_rating', 'int'),
 ('moves', 'string'),
 ('opening_eco', 'string'),
 ('opening_name', 'string'),
 ('opening_ply', 'int'),
 ('increment1', 'int'),
 ('increment2', 'int')]
```

Let's examine the rated column

In [19]:
```
df.groupBy('rated').count().show()
```

```
+-----+-----+
|rated|count|
+-----+-----+
|FALSE| 1852|
|False| 1960|
| TRUE| 7418|
| True| 8399|
+-----+-----+
```

This column needs cleaning, which is performed in the cell below

In [20]:
```
df = df.withColumn("rated", \
              when(df["rated"] == "FALSE", "False").otherwise(df["rated"]))

df = df.withColumn("rated", \
              when(df["rated"] == "TRUE", "True").otherwise(df["rated"]))
```

Let's review the updated column

In [21]:
```
df.groupBy('rated').count().show()
```

```
+-----+-----+
|rated|count|
+-----+-----+
|False| 3812|
| True|15817|
+-----+-----+
```

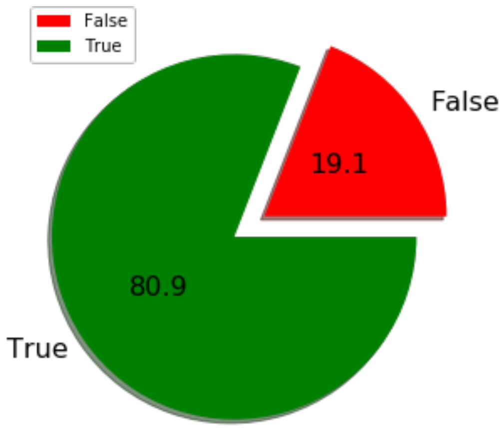
Let's redrop duplicates after those transformations, to catch other possible identical rows

In [22]:
```
print(df.count())
df = df.dropDuplicates()
print(df.count())
```

```
19629
19113
```

Let's visualize the rated column

In [104]:
```
df.groupBy('rated').count().toPandas().plot.pie(y="count", figsize=(5, 5),
                                                 labels=["False", "True"], autopc
t='%.1f',
                                                 colors=["r", "g"], pctdistance=0.
5,
                                                textprops={"size":16}, explode=[0.
1,0.1],
                                                shadow = True)
plt.ylabel("")
plt.show()
```

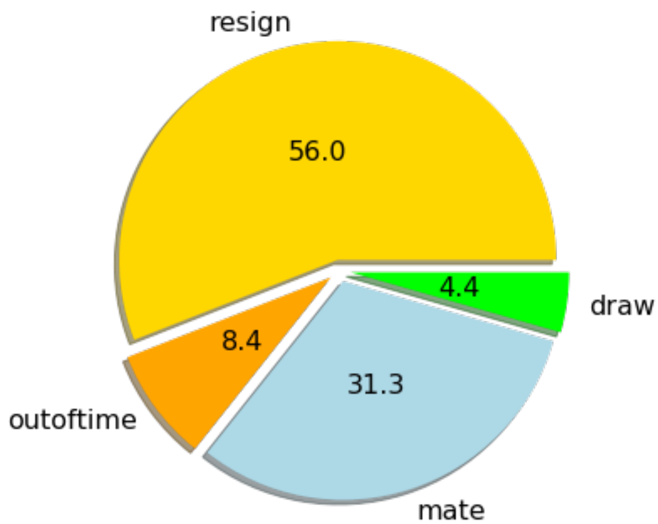


Let's visualize the victory status

In [105]:
```
df.groupBy('victory_status').count().show()
```

```
+--------------+-----+
|victory_status|count|
+--------------+-----+
|        resign|10695|
|     outoftime| 1598|
|          mate| 5974|
|          draw|  846|
+--------------+-----+
```

In [106]:
```
df.groupBy('victory_status').count().toPandas().plot.pie(y="count", figsize=(6, 6),
                                                          labels=["resign", "outoftime", "mate", "draw"], autopct='%.1f',
                                                          colors=["gold", "orange", "lightblue", "lime"], pctdistance=0.5,
                                                         textprops={"size":16}, legend=None, explode=[0.05,0.05,0.05,0.05],
                                                         shadow = True)
plt.ylabel("")
plt.show()
```



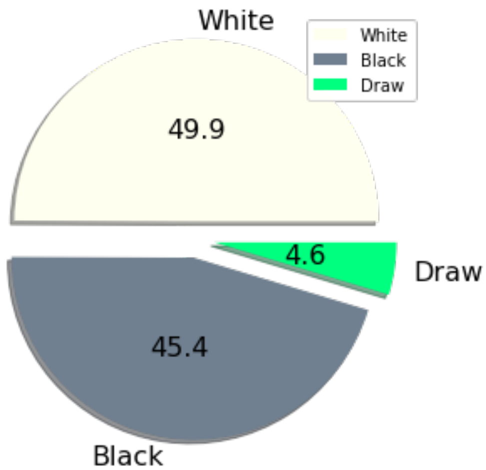
Let's see how winners are distributed: we see that the white player (who moves first) has a slight advantage over the black one;

this is a well-known fact in the chess world

In [107]:
```
df.groupBy('winner').count().show()
```

```
+------+-----+
|winner|count|
+------+-----+
| white| 9545|
| black| 8680|
|  draw|  888|
+------+-----+
```

In [108]:
```
df.groupBy('winner').count().toPandas().plot.pie(y="count", figsize=(5, 5),
                                                  labels=["White", "Black", "Draw"],
autopct='%.1f',
                                                  colors=["ivory", "slategrey", "spr
inggreen"], pctdistance=0.5,
                                                 textprops={"size":16}, explode=[0.
1,0.1, 0.1],
                                                 shadow = True)
plt.ylabel("")
plt.show()
```



We now inspect a bit the column of the moves

In [109]:
```
df.select("moves").show()
```

```
+--------------------+
|               moves|
+--------------------+
|d4 c5 c3 cxd4 cxd...|
|e4 c5 c3 Nc6 d4 c...|
|e4 e5 Nf3 Nc6 d4 ...|
|d4 d5 Nf3 Nf6 Bf4...|
|e4 Nc6 d4 Nf6 e5 ...|
|d4 Nf6 Nc3 d6 e4 ...|
|d4 e6 c4 c5 d5 Qb...|
|d4 f5 e3 Nf6 Nf3 ...|
|d4 d5 c4 c6 Nf3 N...|
|e4 e5 Nf3 Nc6 Bb5...|
|d4 b6 Nf3 Bb7 e3 ...|
|e4 e5 Nf3 d6 Nc3 ...|
|e4 c5 Nf3 d6 d4 c...|
|e4 e5 Nf3 Nc6 Bc4...|
|e4 e5 Nf3 Nc6 Bc4...|
|e4 Nc6 Bc4 Nf6 Nf...|
|d3 e6 e3 Nc6 Nf3 ...|
|e4 Nc6 d4 d5 e5 B...|
|e4 h5 Nf3 g6 d4 e...|
|e4 c5 Nf3 d6 Nc3 ...|
+--------------------+
only showing top 20 rows
```

In [110]:
```
df.select(
        "id",
        f.split("moves", " ").alias("moves"),
        f.posexplode(f.split("moves", " ")).alias("pos", "val")
        ).show()
```

```
+--------+--------------------+---+----+
|      id|               moves|pos| val|
+--------+--------------------+---+----+
|YopBjBqM|[d4, c5, c3, cxd4...|  0|  d4|
|YopBjBqM|[d4, c5, c3, cxd4...|  1|  c5|
|YopBjBqM|[d4, c5, c3, cxd4...|  2|  c3|
|YopBjBqM|[d4, c5, c3, cxd4...|  3|cxd4|
|YopBjBqM|[d4, c5, c3, cxd4...|  4|cxd4|
|YopBjBqM|[d4, c5, c3, cxd4...|  5| Nf6|
|YopBjBqM|[d4, c5, c3, cxd4...|  6| Nc3|
|YopBjBqM|[d4, c5, c3, cxd4...|  7|  g6|
|YopBjBqM|[d4, c5, c3, cxd4...|  8|  b3|
|YopBjBqM|[d4, c5, c3, cxd4...|  9| Bg7|
|YopBjBqM|[d4, c5, c3, cxd4...| 10| Bb2|
|YopBjBqM|[d4, c5, c3, cxd4...| 11| O-O|
|YopBjBqM|[d4, c5, c3, cxd4...| 12|  e3|
|YopBjBqM|[d4, c5, c3, cxd4...| 13|  e6|
|YopBjBqM|[d4, c5, c3, cxd4...| 14| Nf3|
|YopBjBqM|[d4, c5, c3, cxd4...| 15|  a6|
|YopBjBqM|[d4, c5, c3, cxd4...| 16| Be2|
|YopBjBqM|[d4, c5, c3, cxd4...| 17|  d5|
|YopBjBqM|[d4, c5, c3, cxd4...| 18| O-O|
|YopBjBqM|[d4, c5, c3, cxd4...| 19| Nc6|
+--------+--------------------+---+----+
only showing top 20 rows
```

To be able to better analyze our dataset, we first create a new column storing the first two moves and the last two moves, and then split them so that we end up with 4 columns, each for every move that we kept

After some experiments keeping all the moves or just subsets, I ended up choosing the first two and last two as this configuration lead to the best results afterwards, and it was also a good compromise concerning the execution speed of neural nets and the other algorithms

In [23]:
```
#function that retrieves the first two and last two moves from the column moves
def funz_moves(value):
    split_moves = value.split(" ")

    if len(split_moves) >= 4:
        fin = split_moves[:2] + split_moves[-2:]
        fin = " ".join(fin)
        return fin
    else:
        split_moves = " ".join(split_moves)
        return split_moves

udf_funz_moves = udf(funz_moves, StringType())

#creating the new column
df = df.withColumn("moves_new", udf_funz_moves("moves"))
```

In [24]:
```
#let's see some rows of the new dataframe
df.limit(3).toPandas()
```

Out[24]:

	id	rated	turns	victory_status	winner	white_rating	black_rating	moves	opening_eco	opening_na
0	YopBjBqM	True	139	draw	draw	1736	1893	d4 c5 c3 cxd4 cxd4 Nf6 Nc3 g6 b3 Bg7 Bb2 O-O e...	A43	Old Ber Defe
1	SipCSO1G	True	96	resign	black	2209	2044	e4 c5 c3 Nc6 d4 cxd4 cxd4 e6 Nc3 Bb4 Ne2 Nge7 ...	B22	Sici Defe Alapin Varia
2	tpi9ti3J	True	58	resign	black	1652	1624	e4 e5 Nf3 Nc6 d4 exd4 Nxd4 Bb4+ c3 Ba5 Nf5 Qf6...	C45	Scotch Ga Malaı Varia

In [25]:
```
#here we create the new four columns containing the moves we want to keep
df_split2 = df.select(
        "id",
        f.split("moves_new", " ").alias("moves_new"),
        f.posexplode(f.split("moves_new", " ")).alias("pos", "val")
    )\
    .drop("val")\
    .select(
        "id",
        f.concat(f.lit("moves_new"),f.col("pos").cast("string")).alias("name"),
        f.expr("moves_new[pos]").alias("val")
    )\
    .groupBy("id").pivot("name").agg(f.first("val"))
```

In [26]:
```
column_list = ["id"] + ["moves_new{0}".format(i) for i in range(4)]
```

In [27]:
```
df_split2 = df_split2.select(column_list)
```

In [28]:
```
#we join the old dataframe with the new one with the four columns
df = df.join(df_split2, ["id"])
df = df.drop("moves")
```

Let's see the shape of the new dataframe

In [29]:
```
print(df.count())
print(len(df.columns))
```

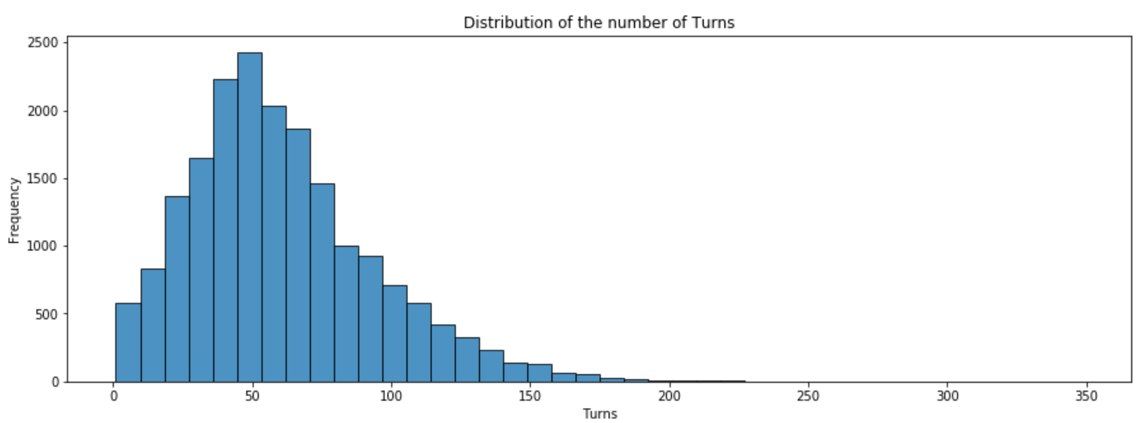
```
19113
17
```

Let's see some summary statistics and a histogram to get some insights from the TURN feature

In [145]:
```
df.select("turns").describe().show()
```

```
+-------+------------------+
|summary|             turns|
+-------+------------------+
|  count|             19113|
|   mean|60.513838748495786|
| stddev| 33.48826396268022|
|    min|                 1|
|    max|               349|
+-------+------------------+
```

In [34]:
```
df.select("turns").toPandas().plot(kind="hist", figsize=(15,5), bins=40, edgecolo
r="black", alpha=0.8, legend=None)
plt.title("Distribution of the number of Turns")
plt.xlabel("Turns")
plt.show()
```

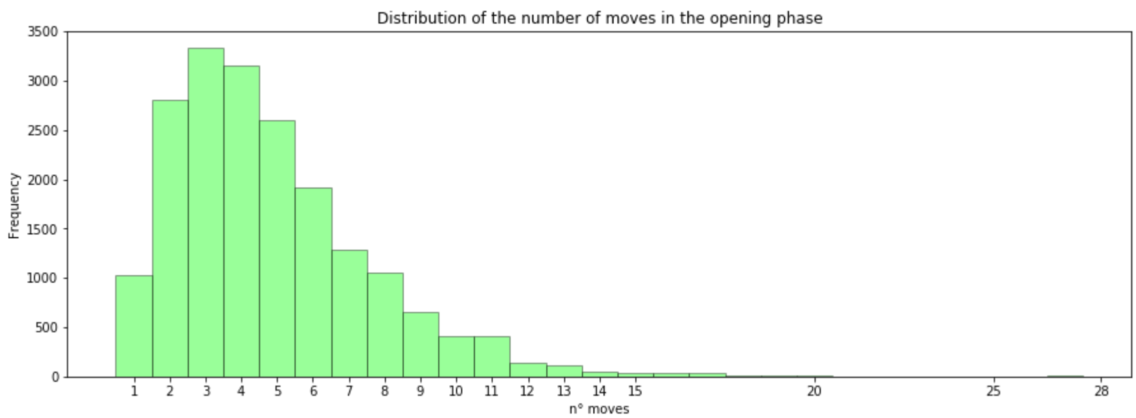


Let's see some summary statistics and a histogram to get some insights from the OPENING_PLAY feature

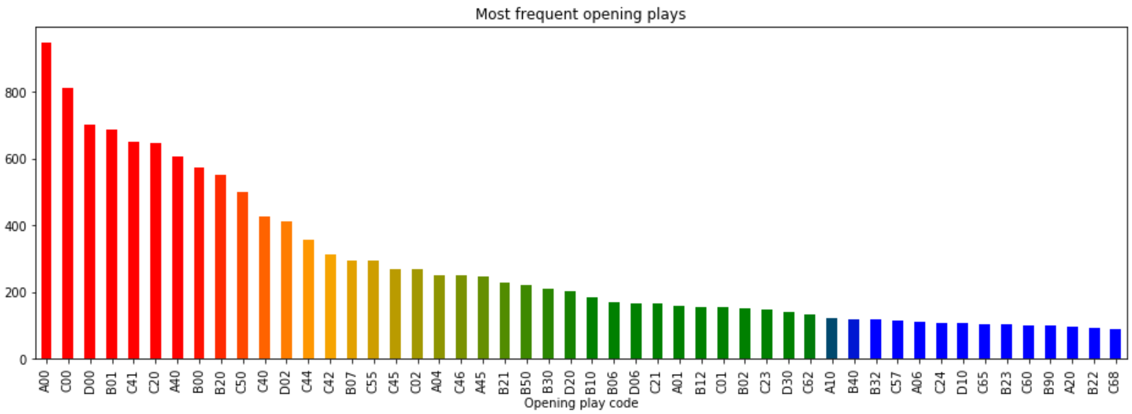
In [35]:
```
df.select("opening_ply").describe().show()
```

```
+-------+------------------+
|summary|       opening_ply|
+-------+------------------+
|  count|             19113|
|   mean| 4.815779835713912|
| stddev|2.7982829080050102|
|    min|                 1|
|    max|                28|
+-------+------------------+
```

In [36]:
```
df.select("opening_ply").toPandas().plot(kind="hist", figsize=(15,5), bins=27, col
or="lime",
                                          edgecolor="black", alpha=0.4, legend=Non
e)
plt.title("Distribution of the number of moves in the opening phase")
plt.xticks([1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5, 10.5,11.5,12.5,13.5,14.5,
15.5,20.5, 25.5,28.5],
           [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,20,25,28])
plt.xlabel("n° moves")
plt.show()
```



In [37]:
```
import matplotlib.colors as mcolors
clist = [(0, "red"), (0.125, "red"), (0.25, "orange"), (0.5, "green"),
         (0.7, "green"), (0.75, "blue"), (1, "blue")]
rvb = mcolors.LinearSegmentedColormap.from_list("", clist)

N = 50
x = np.arange(N).astype(float)
y = np.random.uniform(0, 5, size=(N,))


df.select("opening_eco").toPandas()["opening_eco"].value_counts().iloc[:N].plot(ki
nd="bar",
                                                                                 f
igsize=(16,5),
                                                                                 c
olor=rvb(x/N) )
plt.title("Most frequent opening plays")
plt.xlabel("Opening play code")
plt.show()
```

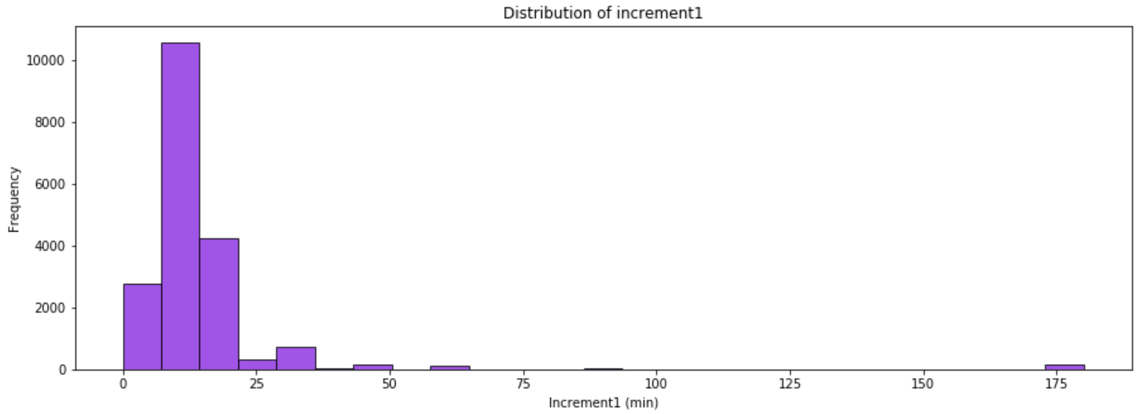


Let's see some summary statistics and a histogram to get some insights from the INCREMENT1 feature

In [38]:
```
df.select("increment1").describe().show()
```

```
+-------+------------------+
|summary|        increment1|
+-------+------------------+
|  count|             19113|
|   mean|13.785277036571967|
| stddev|17.072446733601073|
|    min|                 0|
|    max|               180|
+-------+------------------+
```

In [39]:
```
df.select("increment1").toPandas().plot(kind="hist", figsize=(15,5), color="blueviolet",
                                        bins=25, edgecolor="black", alpha=0.8, legend=None)
plt.title("Distribution of increment1")
plt.xlabel("Increment1 (min)")
plt.show()
```

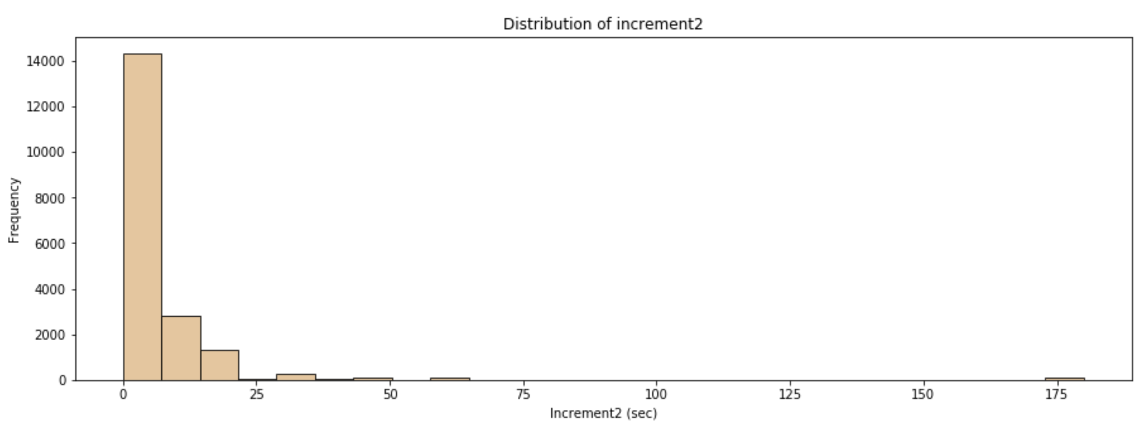


Let's see some summary statistics and a histogram to get some insights from the INCREMENT2 feature

In [40]:
```
df.select("increment2").describe().show()
```

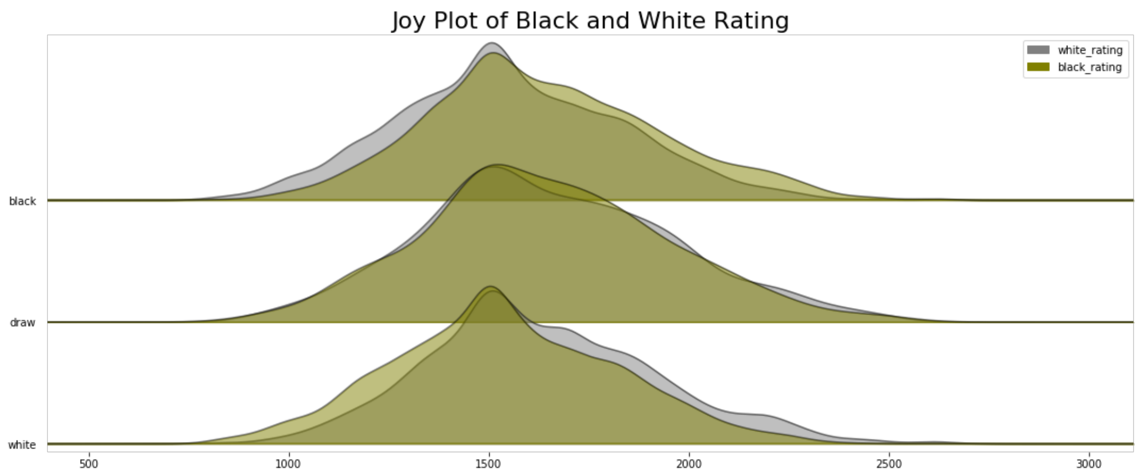
```
+-------+------------------+
|summary|        increment2|
+-------+------------------+
|  count|             19113|
|   mean|5.1464971485376445|
| stddev|13.808620253370899|
|    min|                 0|
|    max|               180|
+-------+------------------+
```

In [41]:
```
df.select("increment2").toPandas().plot(kind="hist", figsize=(15,5), color="burlywood",
                                        bins=25, edgecolor="black", alpha=0.8, legend=None)
plt.title("Distribution of increment2")
plt.xlabel("Increment2 (sec)")
plt.show()
```



Let's explore how the winner columns and the player ratings are related, using joyplots

In [42]:
```
# Draw Plot
fig, axes = joypy.joyplot(df.select("white_rating", "black_rating", "winner").toPa
ndas(),
                          column=['white_rating', 'black_rating'],by = "winner", y
lim='own',
                          color=["grey", "olive"], legend=True,
                          figsize=(15,6), alpha=0.5)

# Decoration

plt.title('Joy Plot of Black and White Rating', fontsize=22)
plt.show()
```



In [146]:
```
df.groupby("winner").mean().select(["winner","avg(white_rating)", "avg(black_ratin
g)"]).show()
```

```
+------+------------------+------------------+
|winner| avg(white_rating)| avg(black_rating)|
+------+------------------+------------------+
| white|1634.7327396542692|1540.4346778418019|
| black|1550.8154377880185| 1639.265668202765|
|  draw|1649.3164414414414|1642.1903153153153|
+------+------------------+------------------+
```

We clearly see from the plots and the box above that there is a correlation between the rating of the players and the winner of the match

We formalize this intuition with the following t-test: we see that p-values are really low when there is a winner, so the difference in rating of players is statistically significant; in case of a draw, the p-value is high, so the difference of player ratings is not statistically significant

In [45]:
```
#create a temp view to perform SQL queries
df.createTempView("dataframe")
```

In [46]:
```
#subsetting player ratings based on the match outcome
df_tt1 = spark.sql("select white_rating, black_rating from dataframe where winner
= '{0}'".format("white")).toPandas()
df_tt2 = spark.sql("select white_rating, black_rating from dataframe where winner
= '{0}'".format("black")).toPandas()
df_tt3 = spark.sql("select white_rating, black_rating from dataframe where winner
= '{0}'".format("draw")).toPandas()
```

In [47]:
```
#performing t-tests to determine if the differences in the means of player ratings
are statistically significant,
#depending on the outcomes
print(scipy.stats.ttest_ind(df_tt1.white_rating, df_tt1.black_rating))
print(scipy.stats.ttest_ind(df_tt2.white_rating, df_tt2.black_rating))
print(scipy.stats.ttest_ind(df_tt3.white_rating, df_tt3.black_rating))
```

```
Ttest_indResult(statistic=22.837035036935784, pvalue=6.591467723433094e-114)
Ttest_indResult(statistic=-20.374237387648417, pvalue=3.2960912165480816e-91)
Ttest_indResult(statistic=0.4918917700218225, pvalue=0.6228567156066165)
```

In the following cells we explore a bit how the columns OPENING_NAME and OPENING_ECO are related

In [50]:
```
ct = pd.crosstab(df.select("opening_eco").toPandas()["opening_eco"],
                 df.select("opening_name").toPandas()["opening_name"])
```

In [51]:
```
ct.head(2)
```

Out[51]:

opening_name	Alekhine Defense	Alekhine Defense #2	Alekhine Defense #3	Alekhine Defense: Balogh Variation	Alekhine Defense: Brooklyn Variation	Alekhine Defense: Exchange Variation	Alekhine Defense: Four Pawns Attack	Alekhine Defense: Four Pawns Attack | 6...Nc6	Alekhine Defense: Four Pawns Attack | Fianchetto Variation	A D
opening_eco										
A00	0	0	0	0	0	0	0	0	0	
A01	0	0	0	0	0	0	0	0	0	

2 rows × 1477 columns

In [52]:
```
list_incongr = []
z = ct.apply(Counter).values
for i, elem in enumerate(z):
    if elem[0]!=364:
        list_incongr.append(i)
```

In [53]:
```
ct2 = ct.iloc[:, list_incongr].T
```

In [54]:
```
list_incongr_col = [i for i in range(365) if ct2.sum().iloc[i]!=0]
```

In [55]:
```
ct3 = ct.iloc[list_incongr_col, list_incongr].T
ct3.head(3)
```

Out[55]:

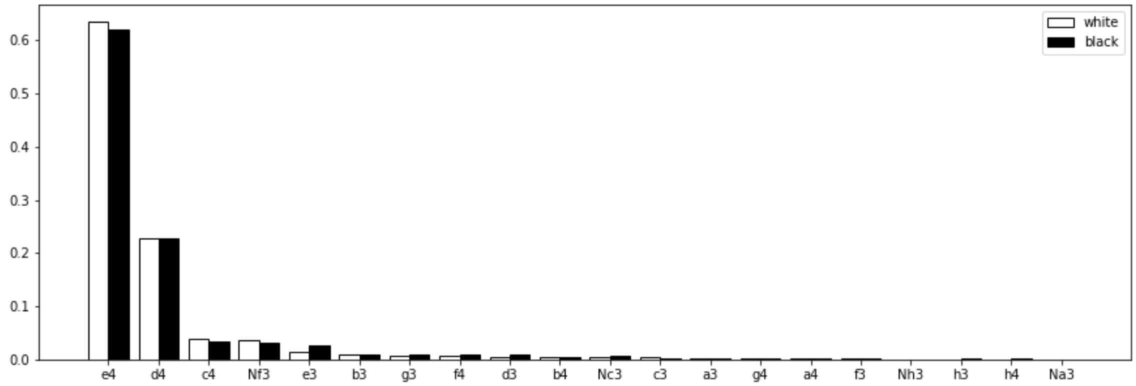
opening_eco opening_name	A02	A03	A04	A06	A07	A08	A09	A20	A21	A34	...	D45	D46	D52	D74	D76	E01	E06
Alekhine Defense	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
Bird Opening	58	11	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
Borg Defense: Borg Gambit	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0

3 rows × 101 columns

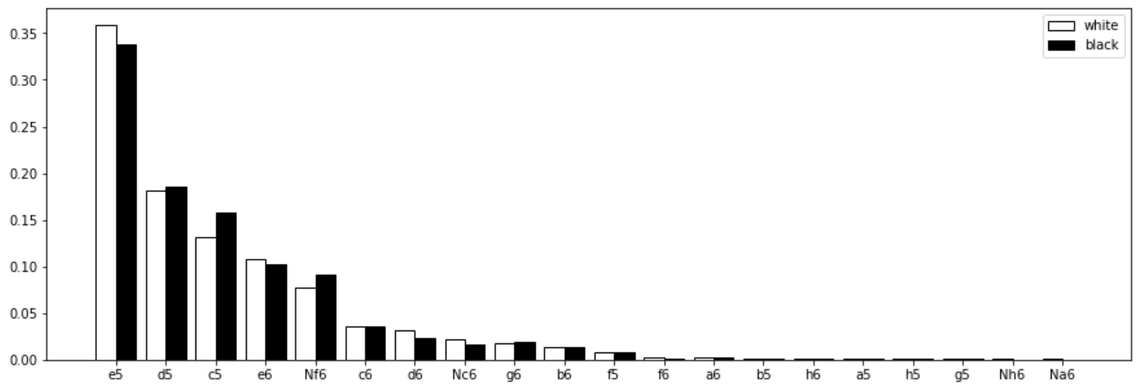
We now draw the barplots of the first two moves of games, to see how they related with the match outcome

We notice in the plots below that the difference in moves with respect to different match outcomes are not strong enough to consider a particular move a good indicator of the final winner

In [147]:
```
#fuction that plots the bars with colors and labels
def subcategorybar(X, vals, width=0.8):
    plt.figure(figsize=(15,5))
    n = len(vals)
    _X = np.arange(len(X))
    labels=["white", "black"]
    for i in range(n):
        plt.bar(_X - width/2. + i/float(n)*width, vals[i]/vals[i].sum(),
                width=width/float(n), align="edge", label=labels[i],
                color=labels[i], edgecolor="black")
    plt.xticks(_X, X)
    plt.legend()

# function that provides the count of the moves divide by match outcome and integr
ates them to solve
# compatibily issues deriving from moves that are never used by a white or black o
pponent
def moves_win(name_col):
    df_pd = df.select(name_col, "winner").toPandas()

    X = list(df_pd[df_pd.winner == "white"]["{0}".format(name_col)].value_counts
().index)
    X2 = list(df_pd[df_pd.winner == "black"]["{0}".format(name_col)].value_counts
().index)

    list_uncommon1 = [i for i in X if i not in X2]
    list_uncommon2 = [i for i in X2 if i not in X]

    X_mod = X + list_uncommon2

    if len(list_uncommon2) == 0:

        Y = df_pd[df_pd.winner == "white"]["{0}".format(name_col)].value_counts().
reindex(X_mod).values

    else:

        for i in range(len(list_uncommon2)):
            Y = df_pd[df_pd.winner == "white"]["{0}".format(name_col)].\
            value_counts().append(pd.Series([0], [list_uncommon2[i]]))

        Y = Y.reindex(X_mod).values

    if len(list_uncommon1) == 0:

        Z = df_pd[df_pd.winner == "black"]["{0}".format(name_col)].value_counts().
reindex(X_mod).values

    else:

        for i in range(len(list_uncommon1)):
            Z = df_pd[df_pd.winner == "black"]["{0}".format(name_col)].\
            value_counts().append(pd.Series([0], [list_uncommon1[i]]))

        Z = Z.reindex(X_mod).values

    subcategorybar(X_mod, [Y,Z])

    plt.show()

moves_win("moves_new0")
```



In [148]:
```
moves_win("moves_new1")
```



In the following boxes we clearly see that the number of turns played is strongly related both to the outcome of a match and to the victory_status

In particular it is clear that matches ending in a draw are characterized by a high number of turns played

In [149]:
```
df.groupby("winner").avg().select("winner", "avg(opening_ply)", "avg(turns)").show
()
```

```
+------+-----------------+------------------+
|winner| avg(opening_ply)|        avg(turns)|
+------+-----------------+------------------+
| white|4.849449973808277| 57.77852278679937|
| black|4.748271889400922|60.795852534562215|
|  draw|5.113738738738738| 87.15878378378379|
+------+-----------------+------------------+
```

In [150]:
```
df.groupby("victory_status").avg().select("victory_status", "avg(opening_ply)", "a
vg(turns)").show()
```

```
+--------------+-----------------+-----------------+
|victory_status| avg(opening_ply)|       avg(turns)|
+--------------+-----------------+-----------------+
|        resign|4.993174380551659|53.91182795698925|
|     outoftime|4.677096370463079|72.97872340425532|
|          mate|4.492132574489454|65.56896551724138|
|          draw|5.120567375886525|84.73404255319149|
+--------------+-----------------+-----------------+
```

We drop other columns that are not needed for the analysis

In [30]:
```
df = df.drop("id", "moves_new", "opening_eco")
```

We notice that the moves_new1, moves_new2 and moves_new3 columsn have some NaNs deriving from that fact that some games terminated after the first, second or third turn; we fill NaN values with 0 to avoid problems in the following phase of modeling; so 0 corresponds to "NO_MOVE", and is an acceptable and consistent value to assign to those columns

In [31]:
```
#check for NaNs and Nulls in every column of df
df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.column
s]).show()
```

```
+-----+-----+--------------+------+------------+------------+------------+------
-----+----------+----------+----------+----------+----------+----------+
|rated|turns|victory_status|winner|white_rating|black_rating|opening_name|openin
g_ply|increment1|increment2|moves_new0|moves_new1|moves_new2|moves_new3|
+-----+-----+--------------+------+------------+------------+------------+------
-----+----------+----------+----------+----------+----------+----------+
|    0|    0|             0|     0|           0|           0|           0|
0|         0|         0|         0|        17|       194|       276|
+-----+-----+--------------+------+------------+------------+------------+------
-----+----------+----------+----------+----------+----------+----------+
```

In [32]:
```
#filling null values with 0
df = df.fillna({'moves_new1':0, 'moves_new2':0, 'moves_new3':0})
```

Feature Engineering and Data Preprocessing

In [33]:
```
#let's see what columns we are left with
df.columns
```

Out[33]:
```
['rated',
 'turns',
 'victory_status',
 'winner',
 'white_rating',
 'black_rating',
 'opening_name',
 'opening_ply',
 'increment1',
 'increment2',
 'moves_new0',
 'moves_new1',
 'moves_new2',
 'moves_new3']
```

We create here a pipeline to preprocess our columns: on categorical features we apply StringIndexer and OHE, whereas we apply only StringIndexer on the label column; finally, we apply VectorAssembler on all the features

In [34]:
```
# Spark Pipeline

#categorical features
cat_features = ['rated', 'opening_name'] + ["moves_new{0}".format(i) for i in rang
e(4)]

#numerical_features
num_features = ['turns', 'white_rating', 'black_rating', 'opening_ply', 'increment
1', 'increment2']

#label to predict
label = 'winner'

# Pipeline Stages List
stages = []

# Loop for StringIndexer and OHE for Categorical Variables
for features in cat_features:

    # Index Categorical Features
    string_indexer = StringIndexer(inputCol=features, outputCol=features + "_inde
x")

    # One Hot Encode Categorical Features
    encoder = OneHotEncoderEstimator(inputCols=[string_indexer.getOutputCol()],
                                     outputCols=[features + "_class_vec"])
    # Append Pipeline Stages
    stages += [string_indexer, encoder]

# Index Label Feature
label_str_index =  StringIndexer(inputCol=label, outputCol="label_index")

# Assemble or Concat the Categorical Features and Numeric Features
assembler_inputs = [feature + "_class_vec" for feature in cat_features] + num_feat
ures

assembler = VectorAssembler(inputCols=assembler_inputs, outputCol="features")

stages += [label_str_index, assembler]
```

Here we see the details of the stages

In [35]:
```
stages
```

Out[35]:
```
[StringIndexer_844b3ff70bd8,
 OneHotEncoderEstimator_80efe4b2af3a,
 StringIndexer_d7607f841299,
 OneHotEncoderEstimator_85f8980c3412,
 StringIndexer_6aab36eb1dad,
 OneHotEncoderEstimator_694757b6bea3,
 StringIndexer_17d4c89bb983,
 OneHotEncoderEstimator_6b4aca6b8aa9,
 StringIndexer_b6411b8ad663,
 OneHotEncoderEstimator_facc55adf211,
 StringIndexer_efb0c90d30bf,
 OneHotEncoderEstimator_e8a3400e0ce7,
 StringIndexer_ce07482d3ac5,
 VectorAssembler_a2b52d3ebab3]
```

Fitting the pipeline and transforming the dataframe

In [36]:
```
# Set Pipeline
pipeline = Pipeline(stages=stages)
print("pipeline initialized")

# Fit Pipeline to Data
pipeline_model = pipeline.fit(df)
print("model fitted")

# Transform Data using Fitted Pipeline
df_transform = pipeline_model.transform(df)
```

```
pipeline initialized
model fitted
```

Let's see the transformed datasets

In [37]:
```
# Preview Newly Transformed Data
df_transform.limit(5).toPandas()
```

Out[37]:

	rated	turns	victory_status	winner	white_rating	black_rating	opening_name	opening_ply	increment1	inc
0	True	139	draw	draw	1736	1893	Old Benoni Defense	2	10	
1	True	96	resign	black	2209	2044	Sicilian Defense: Alapin Variation	3	8	
2	True	58	resign	black	1652	1624	Scotch Game: Malaniuk Variation	8	6	
3	False	39	resign	white	1812	1820	Queen's Pawn Game: London System	5	10	
4	True	93	resign	white	2217	1970	King's Pawn Game: Nimzowitsch Defense	3	10	

5 rows × 28 columns

In [38]:
```
# Select only 'features' and 'label_index' for Final Dataframe
df_transform_fin = df_transform.select('features','label_index')
df_transform_fin.limit(5).toPandas()
```

Out[38]:

	features	label_index
0	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	2.0
1	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	1.0
2	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	1.0
3	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	0.0
4	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	0.0

Let's split the dataset to train models

In [39]:
```
# Split Data into Train / Test Sets
train_data, test_data = df_transform_fin.randomSplit([.8, .2],seed=42)
```

In [40]:
```
train_data.show(5)
```

```
+--------------------+-----------+
|            features|label_index|
+--------------------+-----------+
|(4952,[0,3,1477,1...|        0.0|
|(4952,[0,9,1477,1...|        1.0|
|(4952,[0,9,1477,1...|        1.0|
|(4952,[0,9,1477,1...|        1.0|
|(4952,[0,10,1478,...|        0.0|
+--------------------+-----------+
only showing top 5 rows
```

Model Definition, Training and Evaluation

Machine Learning Models

Decision Tree

Let's first try with a simple Decision Tree, using 5 as maxDepth and Gini as impurity criterion

In [54]:
```
dt = DecisionTreeClassifier(featuresCol = 'features', labelCol = 'label_index',
                            maxDepth = 5, seed=42)
dtModel = dt.fit(train_data)

predictions = dtModel.transform(test_data)
predictions.select('label_index', 'rawPrediction', 'prediction', 'probability').sh
ow(10)
```

```
+-----------+--------------------+----------+--------------------+
|label_index|       rawPrediction|prediction|         probability|
+-----------+--------------------+----------+--------------------+
|        0.0|[1375.0,1715.0,16...|       1.0|[0.42268675069166...|
|        1.0|   [54.0,135.0,12.0]|       1.0|[0.26865671641791...|
|        0.0|[2392.0,1734.0,18...|       0.0|[0.55473098330241...|
|        0.0|   [359.0,78.0,10.0]|       0.0|[0.80313199105145...|
|        0.0|[2392.0,1734.0,18...|       0.0|[0.55473098330241...|
|        0.0|[1375.0,1715.0,16...|       1.0|[0.42268675069166...|
|        2.0|[1375.0,1715.0,16...|       1.0|[0.42268675069166...|
|        0.0|  [966.0,361.0,79.0]|       0.0|[0.68705547652916...|
|        1.0|    [60.0,433.0,9.0]|       1.0|[0.11952191235059...|
|        0.0|[2392.0,1734.0,18...|       0.0|[0.55473098330241...|
+-----------+--------------------+----------+--------------------+
only showing top 10 rows
```

Performances are not so bad, but there is room for improvement, considering that the model hasn't even been tuned

In [55]:
```
evaluator = MulticlassClassificationEvaluator(
    labelCol="label_index", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Set Accuracy = %g" % (accuracy))
```

```
Test Set Accuracy = 0.59337
```

Random Forest

Let's try with a Random Forest, using 20 trees and 5 as maxDepth

In [58]:
```
dt2 = RandomForestClassifier(featuresCol = 'features', labelCol = 'label_index', n
umTrees=20,
                             maxDepth = 5, seed=42)
dt2Model = dt2.fit(train_data)

predictions2 = dt2Model.transform(test_data)
predictions2.select('label_index', 'rawPrediction', 'prediction', 'probability').s
how(10)
```

```
+-----------+--------------------+----------+--------------------+
|label_index|       rawPrediction|prediction|         probability|
+-----------+--------------------+----------+--------------------+
|        0.0|[9.92118618275944...|       0.0|[0.49605930913797...|
|        1.0|[10.1693235631783...|       0.0|[0.50846617815891...|
|        0.0|[9.92118618275944...|       0.0|[0.49605930913797...|
|        0.0|[10.1693235631783...|       0.0|[0.50846617815891...|
|        0.0|[10.0308165621266...|       0.0|[0.50154082810633...|
|        0.0|[9.92118618275944...|       0.0|[0.49605930913797...|
|        2.0|[9.85599977960717...|       0.0|[0.49279998898035...|
|        0.0|[9.92118618275944...|       0.0|[0.49605930913797...|
|        1.0|[9.92118618275944...|       0.0|[0.49605930913797...|
|        0.0|[10.2961729806778...|       0.0|[0.51480864903389...|
+-----------+--------------------+----------+--------------------+
only showing top 10 rows
```

The accuracy level is lower than before, this could be due to the little number of trees used

In [60]:
```
evaluator2 = MulticlassClassificationEvaluator(
    labelCol="label_index", predictionCol="prediction", metricName="accuracy")
accuracy2 = evaluator2.evaluate(predictions2)
print("Test Set Accuracy = %g" % (accuracy2))
```

```
Test Set Accuracy = 0.548822
```

Let's inspect the importance of every single feature in the classification task. The top15 are shown below, along with their importance score

We see that, of course, the last moves are very important, along with the black player rating

In [89]:
```
#custom function to plot feature importance as dataframe
def ExtractFeatureImp(featureImp, dataset, featuresCol):
    list_extract = []
    for i in dataset.schema[featuresCol].metadata["ml_attr"]["attrs"]:
        list_extract = list_extract + dataset.schema[featuresCol].metadata["ml_att
r"]["attrs"][i]
    varlist = pd.DataFrame(list_extract)
    varlist['score'] = varlist['idx'].apply(lambda x: featureImp[x])
    return(varlist.sort_values('score', ascending = False))

print("Feature Importance according to RandomForestClassifier")
ExtractFeatureImp(dt2Model.featureImportances, df_transform_fin, "features").head
(15)
```

```
Feature Importance according to RandomForestClassifier
```

Out[89]:

	idx	name	score
2900	2894	moves_new3_class_vec_Qxf2#	0.086035
1522	1516	moves_new2_class_vec_Kh8	0.061678
2901	2895	moves_new3_class_vec_Qf2#	0.050262
1526	1520	moves_new2_class_vec_Kg1	0.045928
1533	1527	moves_new2_class_vec_Ke2	0.042275
0	4946	black_rating	0.036453
1546	1540	moves_new2_class_vec_Kd2	0.035300
2898	2892	moves_new3_class_vec_Qf7#	0.035298
1543	1537	moves_new2_class_vec_Kd1	0.031475
1536	1530	moves_new2_class_vec_Ke1	0.031467
1540	1534	moves_new2_class_vec_Kf2	0.030360
1597	1591	moves_new2_class_vec_Rf1	0.025012
1523	1517	moves_new2_class_vec_Kf8	0.022163
1528	1522	moves_new2_class_vec_Kg8	0.020249
1529	1523	moves_new2_class_vec_Kd8	0.019544

Let's increase the number of trees and compare the obtained result with the previous one

In [43]:
```
dt2_bis = RandomForestClassifier(featuresCol = 'features', labelCol = 'label_index
', numTrees=50,
                                  maxDepth = 7, seed=42)
dt2bModel = dt2_bis.fit(train_data)

predictions2b = dt2bModel.transform(test_data)
predictions2b.select('label_index', 'rawPrediction', 'prediction', 'probability').
show(10)
```

```
+-----------+--------------------+----------+--------------------+
|label_index|       rawPrediction|prediction|         probability|
+-----------+--------------------+----------+--------------------+
|        0.0|[24.9062157219945...|       0.0|[0.49812431443989...|
|        1.0|[24.3726724937313...|       0.0|[0.48745344987462...|
|        0.0|[25.1859043151886...|       0.0|[0.50371808630377...|
|        0.0|[25.0519876756169...|       0.0|[0.50103975351233...|
|        0.0|[26.7405012631316...|       0.0|[0.53481002526263...|
|        0.0|[25.0480463304278...|       0.0|[0.50096092660855...|
|        2.0|[24.5780693632626...|       0.0|[0.49156138726525...|
|        0.0|[25.0205572718997...|       0.0|[0.50041114543799...|
|        1.0|[24.5687948063188...|       0.0|[0.49137589612637...|
|        0.0|[26.8260392005576...|       0.0|[0.53652078401115...|
+-----------+--------------------+----------+--------------------+
only showing top 10 rows
```

The improvement is slight, so new alternatives are to be preferred over this model

In [44]:
```
evaluator2b = MulticlassClassificationEvaluator(
    labelCol="label_index", predictionCol="prediction", metricName="accuracy")
accuracy2b = evaluator2b.evaluate(predictions2b)
print("Test Set Accuracy = %g" % (accuracy2b))
```

```
Test Set Accuracy = 0.570319
```

Logistic Regression

Let's use the famous logistic regression, in the multiclass version

In [63]:
```
dt3 = LogisticRegression(featuresCol = 'features', labelCol = 'label_index', maxIt
er = 10)

dt3Model = dt3.fit(train_data)

predictions3 = dt3Model.transform(test_data)
predictions3.select('label_index', 'rawPrediction', 'prediction', 'probability').s
how(10)
```

```
+-----------+--------------------+----------+--------------------+
|label_index|       rawPrediction|prediction|         probability|
+-----------+--------------------+----------+--------------------+
|        0.0|[0.07832550896203...|       1.0|[0.35426958643812...|
|        1.0|[-0.7747465818665...|       1.0|[0.01446011173150...|
|        0.0|[2.32551393344361...|       0.0|[0.91492911171162...|
|        0.0|[2.44407460467075...|       0.0|[0.79324394198562...|
|        0.0|[6.27017707591990...|       0.0|[0.99983544741904...|
|        0.0|[-2.1241437294911...|       1.0|[3.04282149222622...|
|        2.0|[1.82497119028347...|       0.0|[0.69026548653943...|
|        0.0|[2.20055334692985...|       0.0|[0.92918447626717...|
|        1.0|[1.14902753173228...|       1.0|[0.10971668049453...|
|        0.0|[0.68414187201709...|       1.0|[0.36142987688250...|
+-----------+--------------------+----------+--------------------+
only showing top 10 rows
```

Without having been tuned, this model considerably outperforms the preceding ones, with an astonishing 70% accuracy on the test set

In [64]:
```
evaluator3 = MulticlassClassificationEvaluator(
    labelCol="label_index", predictionCol="prediction", metricName="accuracy")
accuracy3 = evaluator3.evaluate(predictions3)
print("Test Set Accuracy = %g" % (accuracy3))
```

```
Test Set Accuracy = 0.699042
```

Given the promising results, let's implement a GridSearch with CrossValidation in order to further improve the model

Here we define the parameters that we want to tune using a GridSearch approach

In [68]:
```
#paramGrid = ParamGridBuilder()\
#    .addGrid(dt3.aggregationDepth,[2,5,10])\
#    .addGrid(dt3.elasticNetParam,[0.0, 0.5, 1.0])\
#    .addGrid(dt3.fitIntercept,[False, True])\
#    .addGrid(dt3.maxIter,[10, 100, 1000])\
#    .addGrid(dt3.regParam,[0.01, 0.5, 2.0]) \
#    .build()

paramGrid = ParamGridBuilder()\
    .addGrid(dt3.aggregationDepth,[2,5])\
    .addGrid(dt3.elasticNetParam,[0.0, 0.5, 1.0])\
    .addGrid(dt3.regParam,[0.01, 0.5, 2.0]) \
    .build()
```

We now perform the GridSearch with CrossValidation

In [69]:
```
#it took 1h20min

#Create 3-fold CrossValidator
cv = CrossValidator(estimator=dt3, estimatorParamMaps=paramGrid, evaluator=evaluat
or3, numFolds=3)

# Run cross validations
cvModel = cv.fit(train_data)
print("Model fitted")


predict_train=cvModel.transform(train_data)
predict_test=cvModel.transform(test_data)
print("The accuracy on the training set after cv is {}".format(evaluator3.evaluate
(predict_train)))
print("The accuracy on the test set after cv is {}".format(evaluator3.evaluate(pre
dict_test)))
```

```
Exception ignored in: <object repr() failed>
Traceback (most recent call last):
  File "/usr/local/Cellar/apache-spark/2.4.3/libexec/python/pyspark/ml/wrapper.p
y", line 40, in __del__
    if SparkContext._active_spark_context and self._java_obj is not None:
AttributeError: 'LogisticRegression' object has no attribute '_java_obj'

Model fitted
The accuracy on the training set after cv is 0.7899291896144768
The accuracy on the test set after cv is 0.7311577311577312
```

A further improvement of more than 3% has been obtained

Let's see the best parameters that contributed to this accuracy score

In [77]:
```
bestmod = cvModel.bestModel
bestParams = bestmod.extractParamMap()
# best are aggregationDepth=2, elasticNetParam=0.5, regParam=0.01
```

In [78]:
```
bestParams
```

Out[78]:
```
{Param(parent='LogisticRegression_38de24403f56', name='aggregationDepth', doc='s
uggested depth for treeAggregate (>= 2)'): 2,
 Param(parent='LogisticRegression_38de24403f56', name='elasticNetParam', doc='th
e ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an
L2 penalty. For alpha = 1, it is an L1 penalty'): 0.5,
 Param(parent='LogisticRegression_38de24403f56', name='family', doc='The name of
family which is a description of the label distribution to be used in the model.
Supported options: auto, binomial, multinomial.'): 'auto',
 Param(parent='LogisticRegression_38de24403f56', name='featuresCol', doc='featur
es column name'): 'features',
 Param(parent='LogisticRegression_38de24403f56', name='fitIntercept', doc='wheth
er to fit an intercept term'): True,
 Param(parent='LogisticRegression_38de24403f56', name='labelCol', doc='label col
umn name'): 'label_index',
 Param(parent='LogisticRegression_38de24403f56', name='maxIter', doc='maximum nu
mber of iterations (>= 0)'): 10,
 Param(parent='LogisticRegression_38de24403f56', name='predictionCol', doc='pred
iction column name'): 'prediction',
 Param(parent='LogisticRegression_38de24403f56', name='probabilityCol', doc='Col
umn name for predicted class conditional probabilities. Note: Not all models out
put well-calibrated probability estimates! These probabilities should be treated
as confidences, not precise probabilities'): 'probability',
 Param(parent='LogisticRegression_38de24403f56', name='rawPredictionCol', doc='r
aw prediction (a.k.a. confidence) column name'): 'rawPrediction',
 Param(parent='LogisticRegression_38de24403f56', name='regParam', doc='regulariz
ation parameter (>= 0)'): 0.01,
 Param(parent='LogisticRegression_38de24403f56', name='standardization', doc='wh
ether to standardize the training features before fitting the model'): True,
 Param(parent='LogisticRegression_38de24403f56', name='threshold', doc='threshol
d in binary classification prediction, in range [0, 1]'): 0.5,
 Param(parent='LogisticRegression_38de24403f56', name='tol', doc='the convergenc
e tolerance for iterative algorithms (>= 0)'): 1e-06}
```

Let's try again to get something more out the GridSearch, using the parameters found earlier and trying to tune the others (we split the GridSearch due to time constraints, but it should have been done in one single step)

In [43]:
```
paramGrid2 = ParamGridBuilder()\
    .addGrid(dt3.aggregationDepth,[2])\
    .addGrid(dt3.elasticNetParam,[0.5])\
    .addGrid(dt3.fitIntercept,[False, True])\
    .addGrid(dt3.maxIter,[10, 100])\
    .addGrid(dt3.regParam,[0.01]) \
    .build()
```

We manage to further improve our model, reaching an accuracy of 74% on the test set

In [44]:
```
#it took 43min

#Create 3-fold CrossValidator
cv2 = CrossValidator(estimator=dt3, estimatorParamMaps=paramGrid2, evaluator=evaluator3, numFolds=3)

# Run cross validations
cvModel2 = cv2.fit(train_data)
print("Model fitted")


predict_train2=cvModel2.transform(train_data)
predict_test2=cvModel2.transform(test_data)
print("The accuracy on the training set after cv is {}".format(evaluator3.evaluate(predict_train2)))
print("The accuracy on the test set after cv is {}".format(evaluator3.evaluate(predict_test2)))
```

```
Model fitted
The accuracy on the training set after cv is 0.7934041437188566
The accuracy on the test set after cv is 0.7402227402227403
```

In [45]:
```
bestmod2 = cvModel2.bestModel
bestParams2 = bestmod2.extractParamMap()
# best are aggregationDepth=2, elasticNetParam=0.5, regParam=0.01
# new best are fit_intercept=False, maxIter=100
```

In [46]:
```
bestParams2
```

Out[46]:
```
{Param(parent='LogisticRegression_50a11c3e6bc3', name='aggregationDepth', doc='suggested depth for treeAggregate (>= 2)'): 2,
 Param(parent='LogisticRegression_50a11c3e6bc3', name='elasticNetParam', doc='the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty'): 0.5,
 Param(parent='LogisticRegression_50a11c3e6bc3', name='family', doc='The name of family which is a description of the label distribution to be used in the model. Supported options: auto, binomial, multinomial.'): 'auto',
 Param(parent='LogisticRegression_50a11c3e6bc3', name='featuresCol', doc='features column name'): 'features',
 Param(parent='LogisticRegression_50a11c3e6bc3', name='fitIntercept', doc='whether to fit an intercept term'): False,
 Param(parent='LogisticRegression_50a11c3e6bc3', name='labelCol', doc='label column name'): 'label_index',
 Param(parent='LogisticRegression_50a11c3e6bc3', name='maxIter', doc='maximum number of iterations (>= 0)'): 100,
 Param(parent='LogisticRegression_50a11c3e6bc3', name='predictionCol', doc='prediction column name'): 'prediction',
 Param(parent='LogisticRegression_50a11c3e6bc3', name='probabilityCol', doc='Column name for predicted class conditional probabilities. Note: Not all models output well-calibrated probability estimates! These probabilities should be treated as confidences, not precise probabilities'): 'probability',
 Param(parent='LogisticRegression_50a11c3e6bc3', name='rawPredictionCol', doc='raw prediction (a.k.a. confidence) column name'): 'rawPrediction',
 Param(parent='LogisticRegression_50a11c3e6bc3', name='regParam', doc='regularization parameter (>= 0)'): 0.01,
 Param(parent='LogisticRegression_50a11c3e6bc3', name='standardization', doc='whether to standardize the training features before fitting the model'): True,
 Param(parent='LogisticRegression_50a11c3e6bc3', name='threshold', doc='threshold in binary classification prediction, in range [0, 1]'): 0.5,
 Param(parent='LogisticRegression_50a11c3e6bc3', name='tol', doc='the convergence tolerance for iterative algorithms (>= 0)'): 1e-06}
```

Deep Learning Models

Neural Network without scaling numerical features

Unfortunately we will implement and run some deep neural nets using Pandas because SystemML continued to create problems with Python3, and also switching to Python2 wasn't successful because of other incompatibilities

Let's assign the number of classes, in our case 3 (white winner, black winner, draw), and the input dimension, the number of feature columns after all the transformations

In [1]:
```
# Number of Classes
nb_classes = 3  #train_data.select("label_index").distinct().count()
print("Number of classes:", nb_classes)

# Number of Inputs or Input Dimensions
input_dim = 4956 #len(train_data.select("features").first()[0])  #4953
print("Input dimension:", input_dim)
```

```
Number of classes: 3
Input dimension: 4956
```

After many trials and errors, the following architecture has proven to be the best in predicting the label.

It consists of 5 layers with decreasing number of neurons, 2 l2-regularizers and 5 dropout layers with 0.2 dropout-rate. The activation functions are all ReLu except for the last one, which is a Softmax, the right choice for multiclass classification

I chose Adam as optimizer as a result of trial and error processes, and also because it combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.

Moreover, I chose Accuracy as evaluation metric because the Draw class is very tiny and because it is the most immediate to understand. I experimented also with F1-score and precision, but the information gain was very slight, so I sticked to the basic accuracy

In [141]:
```
# Set up Deep Learning Model / Architecture
model = Sequential()
model.add(Dense(512, input_shape=(input_dim,), activity_regularizer=regularizers.l
2(0.01)))
model.add(Activation('relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(256, activity_regularizer=regularizers.l2(0.01)))
model.add(Activation('relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=["
accuracy"])
```

Let's have a look at the model structure

In [142]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_46 (Dense)             (None, 512)               2537984
_________________________________________________________________
activation_46 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_37 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_47 (Dense)             (None, 256)               131328
_________________________________________________________________
activation_47 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_38 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_48 (Dense)             (None, 256)               65792
_________________________________________________________________
activation_48 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_39 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_49 (Dense)             (None, 128)               32896
_________________________________________________________________
activation_49 (Activation)   (None, 128)               0
_________________________________________________________________
dropout_40 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_50 (Dense)             (None, 3)                 387
_________________________________________________________________
activation_50 (Activation)   (None, 3)                 0
=================================================================
Total params: 2,768,387
Trainable params: 2,768,387
Non-trainable params: 0
_________________________________________________________________
```

We convert the Spark dataframe to a Pandas dataframe, transform the label column with a LabelEncoder, then we drop irrelevant columns as well as the victory_status column, which would falsify the prediction; then we perform OHE. We finally split the dataframe into training set and test set

In [143]:
```
le = LabelEncoder()

dfp = df.toPandas()
col_y = le.fit_transform(dfp.winner)
dfp2 = dfp.drop(["victory_status", "winner", "rated"], 1)
dfp3 = pd.get_dummies(dfp2)
dfp3["winner"] = col_y

X_train, X_test, y_train, y_test = train_test_split(dfp3.drop(["winner"],1),
                                                    dfp3.winner, test_size=0.2, ra
ndom_state=42)
```

Just to see the correspondence between numbers and labels

In [144]:
```
le.inverse_transform([0, 1, 2])
```

Out[144]:
```
array(['black', 'draw', 'white'], dtype=object)
```

Let's have a look at the final dataframe that will be fed to the net

In [145]:
```
dfp.head()
```

Out[145]:

	rated	turns	victory_status	winner	white_rating	black_rating	opening_name	opening_ply	increment1	inc
0	True	139	draw	draw	1736	1893	Old Benoni Defense	2	10	
1	True	96	resign	black	2209	2044	Sicilian Defense: Alapin Variation	3	8	
2	True	58	resign	black	1652	1624	Scotch Game: Malaniuk Variation	8	6	
3	False	39	resign	white	1812	1820	Queen's Pawn Game: London System	5	10	
4	True	93	resign	white	2217	1970	King's Pawn Game: Nimzowitsch Defense	3	10	

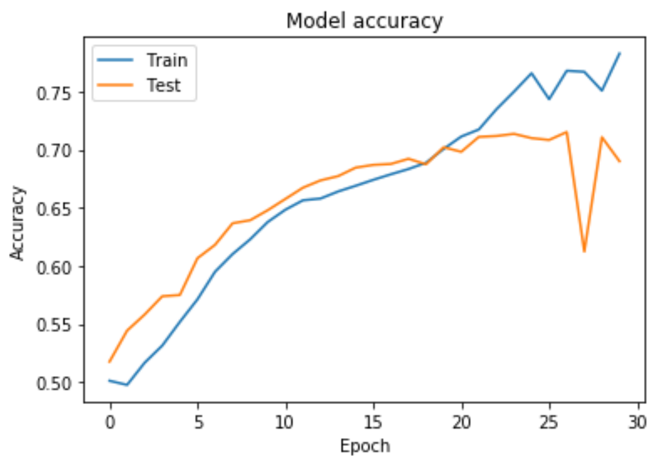
Finally, we train the net for 30 epochs, using a batch size of 256

In [146]:
```
#without StandardScaler
history1 = model.fit(X_train, y_train, epochs=30, batch_size=256, validation_dat
a=(X_test,y_test))
```

```
Train on 15290 samples, validate on 3823 samples
Epoch 1/30
15290/15290 [==============================] - 10s 645us/step - loss: 474516.295
3 - acc: 0.5012 - val_loss: 16100.8306 - val_acc: 0.5177
Epoch 2/30
15290/15290 [==============================] - 7s 468us/step - loss: 11158.7465
- acc: 0.4976 - val_loss: 8310.4972 - val_acc: 0.5443
Epoch 3/30
15290/15290 [==============================] - 7s 434us/step - loss: 6999.5011 -
acc: 0.5167 - val_loss: 5511.2367 - val_acc: 0.5582
Epoch 4/30
15290/15290 [==============================] - 7s 440us/step - loss: 4642.3637 -
acc: 0.5315 - val_loss: 3668.3742 - val_acc: 0.5739
Epoch 5/30
15290/15290 [==============================] - 7s 442us/step - loss: 3111.8474 -
acc: 0.5519 - val_loss: 2472.1366 - val_acc: 0.5749
Epoch 6/30
15290/15290 [==============================] - 7s 447us/step - loss: 2102.4305 -
acc: 0.5712 - val_loss: 1688.1430 - val_acc: 0.6066
Epoch 7/30
15290/15290 [==============================] - 7s 440us/step - loss: 1438.3820 -
acc: 0.5950 - val_loss: 1165.0194 - val_acc: 0.6181
Epoch 8/30
15290/15290 [==============================] - 8s 517us/step - loss: 993.5015 -
acc: 0.6102 - val_loss: 807.3107 - val_acc: 0.6367
Epoch 9/30
15290/15290 [==============================] - 8s 500us/step - loss: 690.6465 -
acc: 0.6229 - val_loss: 565.2326 - val_acc: 0.6393
Epoch 10/30
15290/15290 [==============================] - 8s 546us/step - loss: 481.9598 -
acc: 0.6378 - val_loss: 396.3738 - val_acc: 0.6479
Epoch 11/30
15290/15290 [==============================] - 7s 433us/step - loss: 337.0333 -
acc: 0.6483 - val_loss: 281.7913 - val_acc: 0.6576
Epoch 12/30
15290/15290 [==============================] - 7s 427us/step - loss: 237.3828 -
acc: 0.6564 - val_loss: 198.4267 - val_acc: 0.6673
Epoch 13/30
15290/15290 [==============================] - 6s 415us/step - loss: 167.4914 -
acc: 0.6579 - val_loss: 141.0932 - val_acc: 0.6736
Epoch 14/30
15290/15290 [==============================] - 6s 421us/step - loss: 118.6145 -
acc: 0.6640 - val_loss: 101.3215 - val_acc: 0.6772
Epoch 15/30
15290/15290 [==============================] - 8s 494us/step - loss: 84.9440 - a
cc: 0.6689 - val_loss: 73.0315 - val_acc: 0.6845
Epoch 16/30
15290/15290 [==============================] - 6s 404us/step - loss: 60.9974 - a
cc: 0.6740 - val_loss: 53.2763 - val_acc: 0.6869
Epoch 17/30
15290/15290 [==============================] - 6s 402us/step - loss: 44.3348 - a
cc: 0.6788 - val_loss: 39.4095 - val_acc: 0.6877
Epoch 18/30
15290/15290 [==============================] - 6s 398us/step - loss: 32.8778 - a
cc: 0.6833 - val_loss: 29.4528 - val_acc: 0.6921
Epoch 19/30
15290/15290 [==============================] - 7s 476us/step - loss: 24.6849 - a
cc: 0.6888 - val_loss: 22.7498 - val_acc: 0.6874
Epoch 20/30
15290/15290 [==============================] - 8s 535us/step - loss: 18.8769 - a
cc: 0.7006 - val_loss: 17.5699 - val_acc: 0.7021
Epoch 21/30
15290/15290 [==============================] - 9s 566us/step - loss: 15.0199 - a
cc: 0.7112 - val_loss: 15.2910 - val_acc: 0.6981
Epoch 22/30
```

Let's visualize the model accuracy in the training phase: we see that both the training and test accuracy improve till the 26th epoch, then the test accuracy starts to drop

In [147]:
```
# Plot training & validation accuracy values
plt.plot(history1.history['acc'])
plt.plot(history1.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



We reach a 69% accuracy on the test set

In [148]:
```
model.evaluate(X_test, y_test)
```

```
3823/3823 [==============================] - 1s 212us/step
```

Out[148]:
```
[1.619167909677073, 0.6900340047395264]
```

Neural Network after scaling numerical features

We now scale the numerical features and see how this modification impacts the model performance, and repeat the previous transformations

In [123]:
```
#le = LabelEncoder()
scaler = StandardScaler()

dfp = df.toPandas()
col_y = le.fit_transform(dfp.winner)

#scaling and substituting
for col_name in ["turns", "white_rating", "black_rating", "opening_ply", "incremen
t1", "increment2"]:
    dfp[col_name]= scaler.fit_transform(dfp[col_name].values.reshape(-1, 1))

dfp2 = dfp.drop(["victory_status", "winner", "rated"], 1)
dfp3 = pd.get_dummies(dfp2)
dfp3["winner"] = col_y

X_train, X_test, y_train, y_test = train_test_split(dfp3.drop(["winner"],1),
                                                    dfp3.winner, test_size=0.2, ra
ndom_state=42)
```

Let's see the scaled dataframe

In [124]:
```
dfp.head()
```

Out[124]:

	rated	turns	victory_status	winner	white_rating	black_rating	opening_name	opening_ply	increment1
0	True	2.343753	draw	draw	0.478251	1.043109	Old Benoni Defense	-1.006279	-0.221724
1	True	1.059687	resign	black	2.109200	1.563021	Sicilian Defense: Alapin Variation	-0.648908	-0.338875
2	True	-0.075068	resign	black	0.188611	0.116909	Scotch Game: Malaniuk Variation	1.137949	-0.456026
3	False	-0.642446	resign	white	0.740306	0.791761	Queen's Pawn Game: London System	0.065835	-0.221724
4	True	0.970101	resign	white	2.136785	1.308230	King's Pawn Game: Nimzowitsch Defense	-0.648908	-0.221724

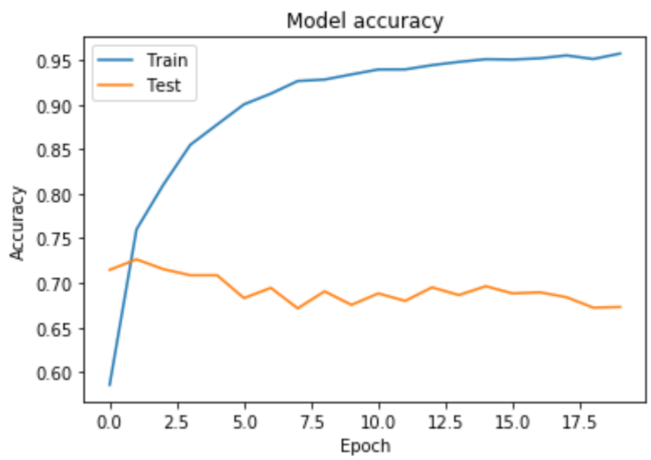
Let's train it, this time for 20 epochs, with a batch size of 128

In [135]:
```
#with StandardScaler
history2 = model.fit(X_train, y_train, epochs=20, batch_size=128, validation_dat
a=(X_test,y_test))
```

```
Train on 15290 samples, validate on 3823 samples
Epoch 1/20
15290/15290 [==============================] - 13s 863us/step - loss: 1.5176 - a
cc: 0.5857 - val_loss: 0.9866 - val_acc: 0.7146
Epoch 2/20
15290/15290 [==============================] - 10s 673us/step - loss: 0.9287 - a
cc: 0.7600 - val_loss: 1.0151 - val_acc: 0.7264
Epoch 3/20
15290/15290 [==============================] - 11s 691us/step - loss: 0.8530 - a
cc: 0.8101 - val_loss: 1.1182 - val_acc: 0.7154
Epoch 4/20
15290/15290 [==============================] - 12s 755us/step - loss: 0.8160 - a
cc: 0.8547 - val_loss: 1.2339 - val_acc: 0.7086
Epoch 5/20
15290/15290 [==============================] - 13s 858us/step - loss: 0.7902 - a
cc: 0.8776 - val_loss: 1.3570 - val_acc: 0.7086
Epoch 6/20
15290/15290 [==============================] - 12s 801us/step - loss: 0.7404 - a
cc: 0.9002 - val_loss: 1.3812 - val_acc: 0.6830
Epoch 7/20
15290/15290 [==============================] - 10s 685us/step - loss: 0.7279 - a
cc: 0.9122 - val_loss: 1.4471 - val_acc: 0.6945
Epoch 8/20
15290/15290 [==============================] - 9s 620us/step - loss: 0.6892 - ac
c: 0.9264 - val_loss: 1.5650 - val_acc: 0.6715
Epoch 9/20
15290/15290 [==============================] - 10s 669us/step - loss: 0.6860 - a
cc: 0.9280 - val_loss: 1.5555 - val_acc: 0.6906
Epoch 10/20
15290/15290 [==============================] - 12s 762us/step - loss: 0.6593 - a
cc: 0.9337 - val_loss: 1.5420 - val_acc: 0.6754
Epoch 11/20
15290/15290 [==============================] - 11s 731us/step - loss: 0.6345 - a
cc: 0.9393 - val_loss: 1.6545 - val_acc: 0.6882
Epoch 12/20
15290/15290 [==============================] - 11s 726us/step - loss: 0.6424 - a
cc: 0.9394 - val_loss: 1.6213 - val_acc: 0.6798
Epoch 13/20
15290/15290 [==============================] - 11s 732us/step - loss: 0.6309 - a
cc: 0.9441 - val_loss: 1.5878 - val_acc: 0.6950
Epoch 14/20
15290/15290 [==============================] - 11s 698us/step - loss: 0.6067 - a
cc: 0.9478 - val_loss: 1.6762 - val_acc: 0.6864
Epoch 15/20
15290/15290 [==============================] - 10s 660us/step - loss: 0.6041 - a
cc: 0.9508 - val_loss: 1.6451 - val_acc: 0.6963
Epoch 16/20
15290/15290 [==============================] - 11s 723us/step - loss: 0.5916 - a
cc: 0.9504 - val_loss: 1.6099 - val_acc: 0.6882
Epoch 17/20
15290/15290 [==============================] - 11s 724us/step - loss: 0.5828 - a
cc: 0.9518 - val_loss: 1.6314 - val_acc: 0.6895
Epoch 18/20
15290/15290 [==============================] - 12s 808us/step - loss: 0.5688 - a
cc: 0.9551 - val_loss: 1.7095 - val_acc: 0.6840
Epoch 19/20
15290/15290 [==============================] - 11s 727us/step - loss: 0.5743 - a
cc: 0.9510 - val_loss: 1.6933 - val_acc: 0.6722
Epoch 20/20
15290/15290 [==============================] - 13s 858us/step - loss: 0.5436 - a
cc: 0.9572 - val_loss: 1.8051 - val_acc: 0.6730
```

Here we see that the training accuracy increases steadily, while the test accuracy starts at a high level and slightly drops during the subsequent epochs, so less epochs are required. We reach 72% of accuracy on the test set

In [136]:
```
# Plot training & validation accuracy values
plt.plot(history2.history['acc'])
plt.plot(history2.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

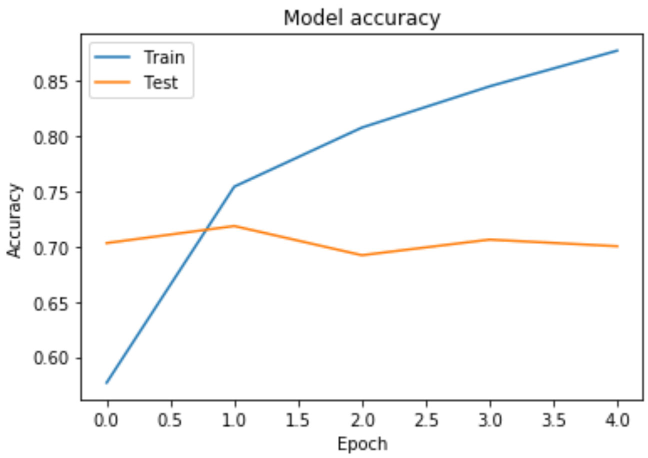


Let's repeat the process here, reducing the number of epochs to just 5

In [138]:
```
#with StandardScaler, less epochs
history3 = model.fit(X_train, y_train, epochs=5, batch_size=128, validation_dat
a=(X_test,y_test))
```

```
Train on 15290 samples, validate on 3823 samples
Epoch 1/5
15290/15290 [==============================] - 16s 1ms/step - loss: 1.4989 - ac
c: 0.5768 - val_loss: 1.0055 - val_acc: 0.7031
Epoch 2/5
15290/15290 [==============================] - 12s 782us/step - loss: 0.9270 - a
cc: 0.7542 - val_loss: 1.0049 - val_acc: 0.7185
Epoch 3/5
15290/15290 [==============================] - 11s 725us/step - loss: 0.8554 - a
cc: 0.8076 - val_loss: 1.1710 - val_acc: 0.6921
Epoch 4/5
15290/15290 [==============================] - 11s 699us/step - loss: 0.8206 - a
cc: 0.8449 - val_loss: 1.1969 - val_acc: 0.7063
Epoch 5/5
15290/15290 [==============================] - 11s 732us/step - loss: 0.7833 - a
cc: 0.8772 - val_loss: 1.3178 - val_acc: 0.7002
```

In [139]:
```
# Plot training validation accuracy values
plt.plot(history3.history['acc'])
plt.plot(history3.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



We now reach 70% accuracy on the test set, so the scaling did NOT improve our results

In [140]:
```
model.evaluate(X_test, y_test)
```

```
3823/3823 [==============================] - 1s 230us/step
```

Out[140]:
```
[0.9364024672463338, 0.7002354172583871]
```

Neural Network with deeper architecture

We try a deeper architecture, with a greater number of trainable parameters and more layers

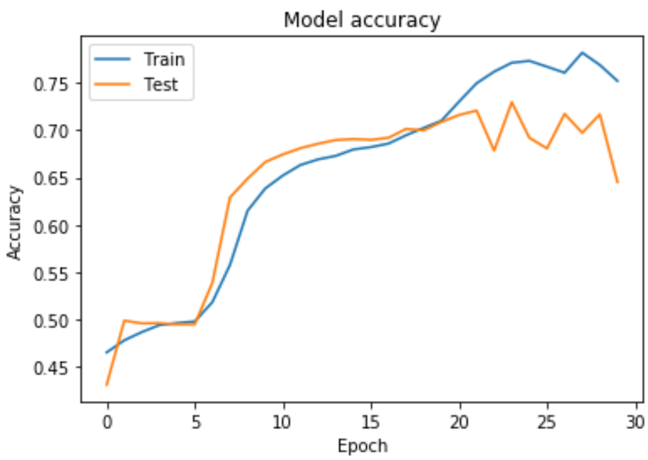
In [159]:
```
model2 = Sequential()
model2.add(Dense(1024, input_shape=(input_dim,), activity_regularizer=regularizers.l2(0.01)))
model2.add(Activation('relu'))
model2.add(Dropout(rate=0.2))
model2.add(Dense(512, activity_regularizer=regularizers.l2(0.01)))
model2.add(Activation('relu'))
model2.add(Dropout(rate=0.2))
model2.add(Dense(256))
model2.add(Activation('relu'))
model2.add(Dropout(rate=0.2))
model2.add(Dense(128))
model2.add(Activation('relu'))
model2.add(Dropout(rate=0.2))
model2.add(Dense(64))
model2.add(Activation('relu'))
model2.add(Dropout(rate=0.2))
model2.add(Dense(nb_classes))
model2.add(Activation('softmax'))
model2.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=["accuracy"])
```

We train the new model with a batch size of 256 for 30 epochs

In [160]:
```
history4 = model2.fit(X_train, y_train, epochs=30, batch_size=256, validation_dat
a=(X_test,y_test))
```

```
Train on 15290 samples, validate on 3823 samples
Epoch 1/30
15290/15290 [==============================] - 19s 1ms/step - loss: 761454.4000
- acc: 0.4653 - val_loss: 28340.7632 - val_acc: 0.4311
Epoch 2/30
15290/15290 [==============================] - 14s 921us/step - loss: 20517.6986
- acc: 0.4780 - val_loss: 15308.5920 - val_acc: 0.4988
Epoch 3/30
15290/15290 [==============================] - 13s 881us/step - loss: 12844.3107
- acc: 0.4868 - val_loss: 9921.8374 - val_acc: 0.4959
Epoch 4/30
15290/15290 [==============================] - 15s 1ms/step - loss: 8283.3197 -
acc: 0.4942 - val_loss: 6419.6704 - val_acc: 0.4962
Epoch 5/30
15290/15290 [==============================] - 15s 998us/step - loss: 5339.0282
- acc: 0.4963 - val_loss: 4171.8381 - val_acc: 0.4949
Epoch 6/30
15290/15290 [==============================] - 14s 944us/step - loss: 3475.9095
- acc: 0.4980 - val_loss: 2737.0189 - val_acc: 0.4949
Epoch 7/30
15290/15290 [==============================] - 13s 857us/step - loss: 2285.2822
- acc: 0.5184 - val_loss: 1811.1994 - val_acc: 0.5388
Epoch 8/30
15290/15290 [==============================] - 13s 853us/step - loss: 1514.8119
- acc: 0.5578 - val_loss: 1211.4257 - val_acc: 0.6291
Epoch 9/30
15290/15290 [==============================] - 14s 912us/step - loss: 1013.3612
- acc: 0.6150 - val_loss: 816.7990 - val_acc: 0.6490
Epoch 10/30
15290/15290 [==============================] - 13s 862us/step - loss: 682.7646 -
acc: 0.6385 - val_loss: 554.2901 - val_acc: 0.6665
Epoch 11/30
15290/15290 [==============================] - 15s 981us/step - loss: 462.9577 -
acc: 0.6523 - val_loss: 378.4063 - val_acc: 0.6746
Epoch 12/30
15290/15290 [==============================] - 15s 989us/step - loss: 315.8344 -
acc: 0.6634 - val_loss: 261.0208 - val_acc: 0.6811
Epoch 13/30
15290/15290 [==============================] - 15s 975us/step - loss: 217.1697 -
acc: 0.6693 - val_loss: 181.1434 - val_acc: 0.6858
Epoch 14/30
15290/15290 [==============================] - 14s 943us/step - loss: 150.1115 -
acc: 0.6730 - val_loss: 128.1626 - val_acc: 0.6898
Epoch 15/30
15290/15290 [==============================] - 15s 953us/step - loss: 105.2546 -
acc: 0.6798 - val_loss: 90.8393 - val_acc: 0.6908
Epoch 16/30
15290/15290 [==============================] - 15s 958us/step - loss: 74.6190 -
acc: 0.6823 - val_loss: 65.4436 - val_acc: 0.6900
Epoch 17/30
15290/15290 [==============================] - 16s 1ms/step - loss: 53.7518 - ac
c: 0.6860 - val_loss: 48.2901 - val_acc: 0.6921
Epoch 18/30
15290/15290 [==============================] - 15s 970us/step - loss: 39.5685 -
acc: 0.6948 - val_loss: 36.4650 - val_acc: 0.7015
Epoch 19/30
15290/15290 [==============================] - 13s 873us/step - loss: 29.8390 -
acc: 0.7026 - val_loss: 28.1016 - val_acc: 0.7000
Epoch 20/30
15290/15290 [==============================] - 13s 882us/step - loss: 23.1054 -
acc: 0.7101 - val_loss: 22.2994 - val_acc: 0.7089
Epoch 21/30
15290/15290 [==============================] - 14s 902us/step - loss: 18.4272 -
acc: 0.7302 - val_loss: 18.2831 - val_acc: 0.7162
Epoch 22/30
```

In [161]:
```
# Plot training validation accuracy values
plt.plot(history4.history['acc'])
plt.plot(history4.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



We reach a worse accuracy, 64.5%, so the higher complexity did NOT provide better results

In [162]:
```
model2.evaluate(X_test, y_test)
```

```
3823/3823 [==============================] - 1s 369us/step
```

Out[162]:
```
[1.838927321648554, 0.6453047345328824]
```

Conclusions

The best result obtained is an accuracy of 74% on the test set which was achieved using a Logistic Regression Model

The results achieved allows us to implement a possible real-time process which, during a game, based on the moves, on the number of turns, on the player ratings etc keeps track of the real-time match outcome probability; when the probability of an outcome goes beyond a certain threshold the platform sends a message to the players suggesting that they should stop the game, thus guaranteeing a high level of entertainment and limiting the boring phases of the game.

Moreover, another application would be to implement a game mode where players start to play from a certain pawn configuration, in order to skip the initial boring phase and to start from a point where the outcome of the game is still uncertain.

Another game mode could consist of starting from a point where one player, usually the more skilled one, starts from a disadvantaged position, to make the match more challenging. The difficulty level can be determined from the outcome probabilities predicted by the model.