2. Problem Statement

In this assignment, students will be using the K-nearest neighbors algorithm to predict how many points NBA players scored in the 2013-2014 season.

A look at the data

Before we dive into the algorithm, let's take a look at our data. Each row in the data contains information on how a player performed in the 2013-2014 NBA season.

Download 'nba_2013.csv' file from this link: https://www.dropbox.com/s/b3nv38jjo5dxcl6/nba_2013.csv?dl=0 (https://www.dropbox.com/s/b3nv38jjo5dxcl6/nba_2013.csv?dl=0)

Here are some selected columns from the data:

- player - name of the player
- pos - the position of the player
- g - number of games the player was in
- gs - number of games the player started
- pts - total points the player scored

There are many more columns in the data, mostly containing information about average player game performance over the course of the season. See this site for an explanation of the rest of them. We can read our dataset in and figure out which columns are present:

Importing and reading the dataset

In [1]:
```
import pandas as pd

#data = pd.read_csv('nba_2013.csv')

#data
nba =''

with open("nba_2013.csv", 'r') as csvfile:
    nba = pd.read_csv(csvfile)
```

```
In [2]: nba
```

Out[2]:

	player	pos	age	bref_team_id	g	gs	mp	fg	fga	fg.	...	drb	trb	ast	stl	blk	tov
0	Quincy Acy	SF	23	TOT	63	0	847	66	141	0.468	...	144	216	28	23	26	30
1	Steven Adams	C	20	OKC	81	20	1197	93	185	0.503	...	190	332	43	40	57	71
2	Jeff Adrien	PF	27	TOT	53	12	961	143	275	0.520	...	204	306	38	24	36	39
3	Arron Afflalo	SG	28	ORL	73	73	2552	464	1011	0.459	...	230	262	248	35	3	146
4	Alexis Ajinca	C	25	NOP	56	30	951	136	249	0.546	...	183	277	40	23	46	63
5	Cole Aldrich	C	25	NYK	46	2	330	33	61	0.541	...	92	129	14	8	30	18
6	LaMarcus Aldridge	PF	28	POR	69	69	2498	652	1423	0.458	...	599	765	178	63	68	123
7	Lavoy Allen	PF	24	TOT	65	2	1072	134	300	0.447	...	192	311	71	24	33	44
8	Ray Allen	SG	38	MIA	73	9	1936	240	543	0.442	...	182	205	143	54	8	84
9	Tony Allen	SG	32	MEM	55	28	1278	204	413	0.494	...	129	208	94	90	19	90
10	Al-Farouq Aminu	SF	23	NOP	80	65	2045	234	494	0.474	...	367	496	114	82	38	88
11	Louis Amundson	PF	31	TOT	19	0	185	16	32	0.500	...	27	55	6	9	11	14
12	Chris Andersen	C	35	MIA	72	0	1396	177	275	0.644	...	250	379	19	32	97	53
13	Alan Anderson	SF	31	BRK	78	26	1773	194	485	0.400	...	135	175	81	48	11	62
14	James Anderson	SG	24	PHI	80	62	2309	309	717	0.431	...	241	300	149	74	28	106
15	Ryan Anderson	PF	25	NOP	22	14	795	155	354	0.438	...	76	142	17	10	7	20
16	Giannis Antetokounmpo	SF	19	MIL	77	23	1897	173	418	0.414	...	261	339	150	60	61	122
17	Carmelo Anthony	PF	29	NYK	77	77	2982	743	1643	0.452	...	477	622	242	95	51	198
18	Joel Anthony	C	31	TOT	33	0	186	12	32	0.375	...	23	38	2	3	12	3
19	Pero Antic	PF	31	ATL	50	26	925	123	294	0.418	...	152	209	58	19	12	55
20	Trevor Ariza	SF	28	WAS	77	77	2723	389	853	0.456	...	376	475	191	126	20	132
21	Hilton Armstrong	C	29	GSW	15	1	97	9	19	0.474	...	28	47	5	4	4	6
22	Darrell Arthur	SF	25	DEN	68	1	1161	162	410	0.395	...	158	210	61	39	47	58
23	Omer Asik	C	27	HOU	48	19	968	101	190	0.532	...	277	378	25	14	37	59
24	D.J. Augustin	PG	26	TOT	71	9	1939	298	718	0.415	...	115	130	313	53	3	125

	player	pos	age	bref_team_id	g	gs	mp	fg	fga	fg.	...	drb	trb	ast	stl	blk	tov
25	Gustavo Ayon	C	28	ATL	26	14	429	52	102	0.510	...	83	125	28	25	10	29
26	Jeff Ayres	PF	26	SAS	73	10	952	101	174	0.580	...	169	258	60	13	25	63
27	Chris Babb	SG	23	BOS	14	0	132	8	30	0.267	...	13	17	3	6	0	3
28	Luke Babbitt	PF	24	NOP	27	2	473	60	154	0.390	...	70	88	29	7	11	15
29	Leandro Barbosa	PG	31	PHO	20	0	368	56	131	0.427	...	32	37	32	7	4	19
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
451	John Wall	PG	23	WAS	82	82	2980	579	1337	0.433	...	295	333	721	149	40	295
452	Gerald Wallace	SF	31	BOS	58	16	1416	116	230	0.504	...	176	212	143	73	14	97
453	Casper Ware	PG	24	PHI	9	0	116	18	42	0.429	...	9	9	10	8	0	5
454	C.J. Watson	PG	29	IND	63	5	1193	146	334	0.437	...	82	101	107	60	8	60
455	Earl Watson	PG	34	POR	24	0	161	3	11	0.273	...	10	15	28	5	1	17
456	Maalik Wayns	PG	22	LAC	2	0	9	1	2	0.500	...	2	2	2	2	0	0
457	Martell Webster	SF	27	WAS	78	13	2157	254	587	0.433	...	184	222	97	41	15	58
458	David West	PF	33	IND	80	80	2472	458	939	0.488	...	422	542	223	61	74	133
459	Russell Westbrook	PG	25	OKC	46	46	1412	346	791	0.437	...	208	263	319	88	7	177
460	D.J. White	PF	27	CHA	2	0	10	0	1	0.000	...	2	2	0	1	0	0
461	Royce White	PF	22	SAC	3	0	9	0	1	0.000	...	0	0	0	0	0	0
462	Deron Williams	PG	29	BRK	64	58	2059	322	716	0.450	...	153	168	392	93	13	143
463	Derrick Williams	SF	22	TOT	78	15	1820	206	482	0.427	...	252	323	56	48	20	76
464	Elliot Williams	SG	24	PHI	67	2	1157	140	337	0.415	...	100	130	72	35	3	68
465	Louis Williams	PG	27	ATL	60	7	1445	197	493	0.400	...	114	124	210	45	4	92
466	Marvin Williams	PF	27	UTA	66	50	1674	231	526	0.439	...	252	334	78	54	31	53
467	Mo Williams	PG	31	POR	74	0	1834	280	672	0.417	...	111	153	321	55	10	149
468	Reggie Williams	SF	27	OKC	3	0	17	5	9	0.556	...	0	0	1	1	0	2
469	Shawne Williams	PF	27	LAL	36	13	751	73	192	0.380	...	142	167	30	19	30	21
470	Jeff Withey	C	23	NOP	58	4	684	69	129	0.535	...	101	150	26	15	50	20

	player	pos	age	bref_team_id	g	gs	mp	fg	fga	fg.	...	drb	trb	ast	stl	blk	tov
471	Nate Wolters	PG	22	MIL	58	31	1309	170	389	0.437	...	116	149	187	35	15	57
472	Metta World Peace	SF	34	NYK	29	1	388	56	141	0.397	...	41	59	17	24	8	19
473	Brandan Wright	C	26	DAL	58	0	1077	224	331	0.677	...	142	244	31	32	55	35
474	Chris Wright	SF	25	MIL	8	0	126	21	35	0.600	...	10	20	5	7	5	5
475	Dorell Wright	SF	28	POR	68	13	984	111	297	0.374	...	162	191	64	23	16	39
476	Tony Wroten	SG	20	PHI	72	16	1765	345	808	0.427	...	159	228	217	78	16	204
477	Nick Young	SG	28	LAL	64	9	1810	387	889	0.435	...	137	166	95	46	12	95
478	Thaddeus Young	PF	25	PHI	79	78	2718	582	1283	0.454	...	310	476	182	167	36	165
479	Cody Zeller	C	21	CHA	82	3	1416	172	404	0.426	...	235	353	92	40	41	87
480	Tyler Zeller	C	24	CLE	70	9	1049	156	290	0.538	...	179	282	36	18	38	60

481 rows × 31 columns

In [3]:
```
# The names of all the columns in the data.
print(nba.columns.values)
```

```
['player' 'pos' 'age' 'bref_team_id' 'g' 'gs' 'mp' 'fg' 'fga' 'fg.' 'x3p'
 'x3pa' 'x3p.' 'x2p' 'x2pa' 'x2p.' 'efg.' 'ft' 'fta' 'ft.' 'orb' 'drb'
 'trb' 'ast' 'stl' 'blk' 'tov' 'pf' 'pts' 'season' 'season_end']
```

Euclidean distance

We can use the principle of euclidean distance to find the most similar NBA players to Lebron James.

In [4]:

```python
# Select Lebron James from our dataset
selected_player = nba[nba["player"] == "LeBron James"].iloc[0]

# Choose only the numeric columns (we'll use these to compute euclidean distance)
distance_columns = ['age', 'g', 'gs', 'mp', 'fg', 'fga', 'fg.', 'x3p', 'x3pa', 'x3p.', 'x2p'

selected_player
```

Out[4]:

```
player          LeBron James
pos                       PF
age                       29
bref_team_id             MIA
g                         77
gs                        77
mp                      2902
fg                       767
fga                     1353
fg.                    0.567
x3p                      116
x3pa                     306
x3p.                0.379085
x2p                      651
x2pa                    1047
x2p.                0.621777
efg.                    0.61
ft                       439
fta                      585
ft.                     0.75
orb                       81
drb                      452
trb                      533
ast                      488
stl                      121
blk                       26
tov                      270
pf                       126
pts                     2089
season             2013-2014
season_end              2013
Name: 225, dtype: object
```

In [5]:
```python
import math

def euclidean_distance(row):
    """
    A simple euclidean distance function
    """
    inner_value = 0
    for k in distance_columns:
        inner_value += (row[k] - selected_player[k]) ** 2
    return math.sqrt(inner_value)

# Find the distance from each player in the dataset to lebron.
lebron_distance = nba.apply(euclidean_distance, axis=1)
lebron_distance
```

Out[5]:
```
0      3475.792868
1              NaN
2              NaN
3      1189.554979
4      3216.773098
5              NaN
6       960.443178
7      3131.071083
8      2326.129199
9      2806.955657
10     2277.933945
11             NaN
12     2819.058890
13     2534.074598
14     1970.085795
15     3262.065464
16     2451.378405
17      485.856006
18             NaN
19     3246.515831
20     1539.172839
21             NaN
22     2969.043638
23             NaN
24     2023.603985
25             NaN
26             NaN
27             NaN
28     3754.041967
29     3835.882699
          ...
451     716.243023
452    2996.450583
453    4135.156714
454    3023.456473
455    4138.570811
456            NaN
457    2206.524879
458    1347.758158
459    2136.309449
460            NaN
461            NaN
462    1922.713718
463    2364.771676
464    3033.755934
465    2625.998112
466    2495.296784
467    2232.354830
```

```
468            NaN
469    3525.434026
470    3574.911070
471    2873.509019
472    3831.629171
473            NaN
474    4124.384593
475    3230.143973
476    1948.158130
477    1851.909840
478     949.668916
479    2699.963932
480    3075.753429
Length: 481, dtype: float64
```

Normalizing columns

Once you have multiple columns, one column may have larger impact than that of others columns becoz of its value, and thus dwarf the impact of racing_stripes values in the euclidean distance calculations.

This can be bad, because a variable having larger values doesn't necessarily make it better at predicting what rows are similar.

A simple way to deal with this is to normalize all the columns to have a mean of 0, and a standard deviation of 1. This will ensure that no single column has a dominant impact on the euclidean distance calculations.

To set the mean to 0, we have to find the mean of a column, then subtract the mean from every value in the column. To set the standard deviation to 1, we divide every value in the column by the standard deviation. The formula is //(x=\frac{x-\mu}{\sigma}).

In [6]:

```
# Select only the numeric columns from the NBA dataset
nba_numeric = nba[distance_columns]

# Normalize all of the numeric columns
nba_normalized = (nba_numeric - nba_numeric.mean()) / nba_numeric.std()
```

Finding the nearest neighbor

We now know enough to find the nearest neighbor of a given row in the NBA dataset. We can use the distance.euclidean function from scipy.spatial, a much faster way to calculate euclidean distance.

In [7]:

```python
from scipy.spatial import distance

# Fill in NA values in nba_normalized
nba_normalized.fillna(0, inplace=True)

# Find the normalized vector for lebron james.
lebron_normalized = nba_normalized[nba["player"] == "LeBron James"]

# Find the distance between lebron james and everyone else.
euclidean_distances = nba_normalized.apply(lambda row: distance.euclidean(row, lebron_normal

# Create a new dataframe with distances.
distance_frame = pd.DataFrame(data={"dist": euclidean_distances, "idx": euclidean_distances.
distance_frame.sort_values("dist", inplace=True)
print(distance_frame)

# Find the most similar player to lebron (the lowest distance to lebron is lebron, the secon
second_smallest = distance_frame.iloc[1]["idx"]
most_similar_to_lebron = nba.loc[int(second_smallest)]["player"]
print(most_similar_to_lebron)
```

```
          dist  idx
225   0.000000  225
17    4.171854   17
136   4.206786  136
128   4.382582  128
185   4.489928  185
133   4.619280  133
123   4.673849  123
162   4.844802  162
332   4.893563  332
451   4.937466  451
160   4.938801  160
179   5.084443  179
423   5.305866  423
218   5.476262  218
197   5.542109  197
307   5.546064  307
416   5.604720  416
277   5.628071  277
110   5.724909  110
272   5.927671  272
85    6.012417   85
450   6.094992  450
278   6.104387  278
99    6.171672   99
253   6.221577  253
478   6.254191  478
347   6.309149  347
177   6.313540  177
345   6.362956  345
3     6.473960    3
..         ...  ...
263  15.560209  263
455  15.610856  455
308  15.658503  308
108  15.667851  108
356  15.715293  356
134  15.735660  134
53   15.736456   53
425  15.750488  425
404  15.850572  404
```

```
431  15.889840  431
222  15.959102  222
327  16.065221  327
321  16.201575  321
324  16.223200  324
424  16.397847  424
224  16.410734  224
339  16.562235  339
271  16.594910  271
63   16.700647   63
226  16.815931  226
46   16.861441   46
109  16.893370  109
460  18.235339  460
190  18.269501  190
461  18.306939  461
219  18.398069  219
388  18.421438  388
210  18.474774  210
351  18.846871  351
240  18.971483  240

[481 rows x 2 columns]
Carmelo Anthony
```

Generating training and testing sets

Now that we know how to find the nearest neighbors, we can make predictions on a test set. We'll try to predict how many points a player scored using the 5 closest neighbors. We'll find neighbors by using all the numeric columns in the dataset to generate similarity scores.

First, we have to generate test and train sets. In order to do this, we'll use random sampling. We'll randomly shuffle the index of the nba dataframe, and then pick rows using the randomly shuffled values.

If we didn't do this, we'd end up predicting and training on the same data set, which would overfit. We could do cross validation also, which would be slightly better, but slightly more complex.

In [8]:
```
import random
from numpy.random import permutation

# Randomly shuffle the index of nba.
random_indices = permutation(nba.index)
# Set a cutoff for how many items we want in the test set (in this case 1/3 of the items)
test_cutoff = math.floor(len(nba)/3)
# Generate the test set by taking the first 1/3 of the randomly shuffled indices.
test = nba.loc[random_indices[1:test_cutoff]]
# Generate the train set with the rest of the data.
train = nba.loc[random_indices[test_cutoff:]]
```

Using sklearn for k nearest neighbors

Sklearn performs the normalization and distance finding automatically, and lets us specify how many neighbors we want to look at.

In [9]:
```
# Taking only  the relevant columns
final_columnns = ['age', 'g', 'gs', 'mp', 'fg', 'fga', 'fg.', 'x3p', 'x3pa', 'x3p.', 'x2p',


# The columns that we will be making predictions with.
x_columns = ['age', 'g', 'gs', 'mp', 'fg', 'fga', 'fg.', 'x3p', 'x3pa', 'x3p.', 'x2p', 'x2pa
# The column that we want to predict.
y_column = ["pts"]
```

In [10]:
```
final_train = train[final_columnns]
#final_train
final_test = test[final_columnns]
```

In [11]:
```
# Finding the persent of rows having atleast onw NaN value - Data Preprocessing
print('Training Data === ', final_train[final_columnns].isnull().T.any().T.sum()*100/final_t
print('Test Data === ', final_test[final_columnns].isnull().T.any().T.sum()*100/final_test[f
```

```
Training Data ===  14.330218068535826 %
Test Data ===  20.12578616352201 %
```

In [12]:
```
# Deleting NaN Rows from the dataset - Data Preprocessing
final_train.dropna( axis=0, inplace = True)
final_test.dropna( axis=0, inplace = True)
```

```
C:\Users\prashant_gupta1\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel_lau
ncher.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexin
g.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas-docs/stable/indexing.htm
l#indexing-view-versus-copy)

C:\Users\prashant_gupta1\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel_lau
ncher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexin
g.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas-docs/stable/indexing.htm
l#indexing-view-versus-copy)
  This is separate from the ipykernel package so we can avoid doing imports until
```

In [13]:
```
# Reconfirming on NaN
print('Training Data === ', final_train[final_columnns].isnull().T.any().T.sum()*100/final_t
print('Test Data === ', final_test[final_columnns].isnull().T.any().T.sum()*100/final_test[f
```

```
Training Data ===  0.0 %
Test Data ===  0.0 %
```

In [14]:
```
print(final_train[x_columns].shape)
print(final_train[y_column].shape)
```

```
(275, 25)
(275, 1)
```

In [15]:

```python
from sklearn.neighbors import KNeighborsRegressor
# Create the knn model.
# Look at the five closest neighbors. We are choosing here 5 neighbours
knn = KNeighborsRegressor(n_neighbors=5)
# Fit the model on the training data.
knn.fit(final_train[x_columns], final_train[y_column])
# Make point predictions on the test set using the fit model.
predictions = knn.predict(final_test[x_columns])

len(predictions)
```

Out[15]:

```
127
```

Computing error

Now that we know our point predictions, we can compute the error involved with our predictions. We can compute mean squared error.

In [16]:

```python
# Get the actual values for the test set.
actual = final_test[y_column]

# Compute the mean squared error of our predictions.
mse = (((predictions - actual) ** 2).sum()) / len(predictions)
```

In [17]:

```python
mse
```

Out[17]:

```
pts    11371.621417
dtype: float64
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: