1. PROJECT

1.1 Dataset

In this project StackOverflow Dataset is used which is located in HDFS at */data/stackoverflow.* The Dataset consists of XML files comprising data of Posts, Users, Badges, Votes, Comments, Postlinks and PostsHistory.

1.2 Objective

The main objective of the project is to implement K-means Clustering algorithm using Python and Spark on HDFS. Following are the goals achieved through clustering:

- Clustering is implemented on User base data to group similar users on the basis of their skills. Their skills are quantified taking appropriate features as discussed in sec 1.3.1

- Also, K-means algorithm is used to make homogeneous clusters of Posts on the basis of their popularity which is determined by taking suitable features, also discussed in sec 1.3.1

- Elbow method is used to identify optimal number of clusters, and machine learning techniques such as normalization and one hot representation is implemented without using mlib library.

- The results are discussed, justified and the performance of the algorithm is evaluated based on output.

- Lastly, Mlib library is used to obtain the outputs for both the cases and the results are compared.

1.3 Project Implementation

1.3.1 Feature Selection

- For the objective of grouping similar Users on the basis of their Skillset & grouping similar posts on the basis of their popularity following features are selected:

Clustering on Users			
Users.xml	**Badges.xml**	**Posts.xml**	**Comments.xml**
Id	UserId	Id	Id
Reputation	Name	PostTypeId	UserId
Views		OwnerUserId	
Upvotes			
DownVotes			
Age			

Clustering on Posts
Posts.xml
PostId
PostTypeId
Tags
Score
ViewCount
AnswerCount
CommentCount
FavoriteCount

1.3.2 Data Retrieval

- Required packages are imported first.

```
import sys
import numpy as np
from pyspark import SparkContext
from pyspark.sql import SQLContext, Row
import xml.etree.ElementTree as ET
import re
from pyspark.sql.functions import stddev_pop, avg, broadcast
```

- Xml Data files are read using .textFile method of sc and a RDD is created

```
users = sc.textFile("/data/stackoverflow/Users", 20)
badges = sc.textFile("/data/stackoverflow/Badges", 20)
posts = sc.textFile("/data/stackoverflow/Posts", 20)
comments = sc.textFile("/data/stackoverflow/Comments", 20)
```

- Due to the limitation of reading large amounts of data in XML format, parsers are designed to read each xml file using ElementTree Package Library of Python.

For User Clustering (Shown for Users.xml):

```
def preProcessUsers(data):
      try:
            root = ET.fromstring(data)
             return [int(root.attrib['Id']),int(root.attrib['Reputation']),
int(root.attrib['Views']),int(root.attrib['UpVotes']),int(root.attrib['DownVotes'])
, int(root.attrib['Age'])]
      except:
            print("Ignoring record")
```

For Post Clustering:

```
def preProcessPosts(data):
      try:
            root = ET.fromstring(data)
             return [int(root.attrib['Id']), int(root.attrib['PostTypeId']),
int(root.attrib['Score']), int(root.attrib['ViewCount']), (root.attrib['Tags']),
int(root.attrib['AnswerCount']), int(root.attrib['CommentCount']),
int(root.attrib['FavoriteCount'])]

      except:
            print("Ignoring record")
```

1.3.3 Pre-Processing Data

- RDD's are filtered removing all the null entries and then converted to dataframes using .toDF method.
- For clustering of Users,
    - Two new attributes are created (comment_count, posts_count) using the .count() method on the comment and posts list.
    - To find the number of posts, posts are first filtered by taking PostsTypeId = 2, thus taking only the answer posted by the user into account.
    - Badges data which is StringType is passed to OneHotFunction (sec 1.3.4)
    - Finally all the dataframes are joined using .join() method

- In case of clustering of Posts,
  - Only PostTypeId =1 is taken, thus taking only questions into account.
  - Tags are filtered into favourable format and 100 most common tags are selected for clustering as it was becoming computationally extremely difficult to take all the tags for clustering.

For User Clustering:

```
usersDF= users.map(preProcessUsers).filter(lambda x: x is not None).toDF(['UserId',
'Reputation', 'Views', 'UpVotes', 'DownVotes', 'Age'])

postsData = posts.map(preProcessPosts).filter(lambda x: x is not None).toDF(['Id',
'PostTypeId', 'UserId'])
filteredPostData = postsData.filter("PostTypeId = 2")
postsDF=filteredPostData.groupBy("UserId").count().withColumnRenamed("count",
"post_count")

commentsData= comments.map(preProcessComments).filter(lambda x: x is not
None).toDF(['Id', 'UserId'])
commentsDF= commentsData.groupBy("UserId").count().withColumnRenamed("count",
"comment_count")

badgesData = badges.map(preProcessBadges).filter(lambda x: x is not None)
badgesDF = oneHotName(badgesData)

finalData=usersDF.join(postsDF,["UserId"],"left_outer").join(commentsDF,["UserId"],
"left_outer").join(badgesDF, ["UserId"], "left_outer").fillna(0)
```

For Post Clustering:

```
postsData = posts.map(preProcessPosts).filter(lambda x: x is not None)
tags = postsData.map(lambda x: x[4]).flatMap(lambda y: y.split("><"))
tagsFiltered1 = tags.map(lambda x: x.replace("<","").replace(">",""))
tagsFiltered2 = tagsFiltered1.map(lambda x: re.sub('[^a-zA-Z0-9+#]', '_', x))
tagsCount = tagsFiltered2.countByValue()
tags100 = dict(Counter(tagsCount).most_common(100))
columns = tags100.keys()
```

1.3.4 One Hot Implementation

- The RDD of Badges after pre-processing is passed to the oneHotName Function, where it is converted to dataframe and distinct badges (name) are selected using .distinct() method and stored in the variable distinctColumn.
- The distinct badges are converted to list and iterated over to filter unwanted characters using regex. The character "." has to be filtered also
- Finally, the Badges RDD is grouped by UserId and mapped over using another function helper which converts column of distinct badges names into one hot notation for each respective UserId.
- The oneHotName function returns the dataframe with UserId and distinct Badges names in One Hot notation as columns

```
def oneHotName(data):
      df = data.toDF(['UserId', 'Name'])
      distinctColumn = df.select("Name").distinct()
      columns_temp = [str(i.Name) for i in distinctColumn.collect()]
      columns = [re.sub('[^a-zA-Z0-9+]', '_', x) for x in columns_temp]
      groupedData = data.groupBy(lambda p: p[0])
      processedData = groupedData.map(lambda p :(p[0], helper(p[1],
columns)))
      finalDF = processedData.map(lambda (key, data): Row(key, *[eachColumn
for eachColumn in data])).toDF(['UserId'] + columns)
      return finalDF

def helper(val, columns):
      a = [0] * len(columns)
      for i in val:
            match = re.sub('[^a-zA-Z0-9+]', '_', i[1])
            a[columns.index(match)] = 1
      return a
```

For Clustering of Posts, Tags are converted to One Hot notation in similar manner.

1.3.5 Normalization

- Since our features are scaled to incomparable variances, the Euclidian distance used in K-means algorithm tend to give bias in the performance. Hence, the normalization is done on the selected features.
- Final Dataframe obtained after joining, and Features that are to be normalized are selected and passed to the normalizeFeatures function.
- Columns that are to be normalized are extracted. UserId and Badges columns are dropped.
- Finally, PySpark sql functions – stddev_pop & avg are used to find standard deviation and average of each column and broadcasted in a column using broadcast function.
- Each feature point is normalized by subtracting the average and dividing with the standard deviation of that particular feature column.
- The function returns the dataframe of all features combining the features that are normalized with the features that are not normalized (UserId, Badges)

```
cols = ["Reputation", "Views", "UpVotes", "DownVotes", "Age", "post_count",
"comment_count"]
      normalizedData = normalizeFeatures(finalData, cols).cache()


def normalizeFeatures(df, cols):
      allCols = df.columns
      _ = [allCols.remove(x) for x in cols]
      stats = (df.groupBy().agg(*([stddev_pop(x).alias(x + '_stddev') for x
in cols] + [avg(x).alias(x + '_avg') for x in cols])))
      df = df.join(broadcast(stats))
      exprs = [x for x in allCols] + [((df[x] - df[x + '_avg']) / df[x +
'_stddev']).alias(x) for x in cols]
      return df.select(*exprs)
```

1.3.6 K-Means Implementation

- UserId feature column is dropped before applying K-means Algorithm.
- Centroids are initialised by randomly selecting data points from RDD using .takeSample
- K is taken as 5 in this case. Most optimal K is obtained using elbow method (Details in sec 1.3.7)
- A variable convergeDist is assigned a very small value to break the loop once the Eucledian distance between the centroids and their respective cluster points becomes lesser than the assigned value.
- Cluster Assignment step of the algorithm is implemented by passing points and centroids to a assignCluster function. The function returns the index of a cluster for each data point to which it is closest to.
- RDD closest is made by mapping key value pairs where key is the output from assignCluster function(closest cluster)and the value is the tuple ofdatapoint and constant one which is used to find the new centroids in the next step.
- Next, Movingcentroid step is implemented in two parts. First a pointStats RDD is made by applying reduceByKey transformation to closest RDD.
  Output after reduceByKey:(clusterIndex, (sumOfPoints, noOfPointsInEachCluster))
- Then the new centroid points are calculated by dividingthe sum of points and with the total number of points for each cluster.
  Outputafter map:(clusterIndex, sumOfPoints/noOfPointsInEachCluster)

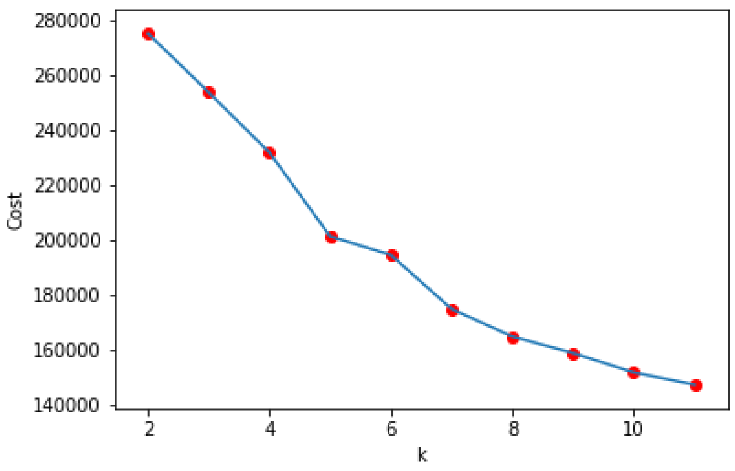
```
data = normalizedData.drop(normalizedData.UserId)

def assignCluster(p, centers):
      bestIndex = 0
        closest = float("+inf")
        for i in range(len(centers)):
            distance = np.sum((np.array(p) - centers[i]) ** 2)
            if distance < closest:
                closest = distance
                bestIndex = i
        return bestIndex


k = 5
convergeDist = float(1e-2)
kPoints = data.rdd.takeSample(False, k, 1)
tempDist = 1.0

while tempDist > convergeDist:
            closest = data.map(lambda p: (assignCluster(p, kPoints),
(np.array(p), 1)))
            pointStats = closest.reduceByKey(lambda p1_c1, p2_c2: (p1_c1[0]
+ p2_c2[0], p1_c1[1] + p2_c2[1]))
            newPoints = pointStats.map(lambda st: (st[0], st[1][0] /
float(st[1][1]))).collect()
            tempDist = sum(np.sum((kPoints[index] - p) ** 2) for (index, p)
in newPoints)
            for (iK, p) in newPoints:
                 kPoints[iK] = p
```

1.3.7 Finding most Optimal K (Elbow Method)

The idea of the elbow method is to run k-means clustering on the dataset for a range of values of k, and for each value of k calculate the sum of squared errors (SSE).
Then, plot a line chart of the SSE for each value of k. If the line chart looks like an arm, then the "elbow" on the arm is the value of k that is the best.

In this project, values from K = 2 to 11 are plotted for against their cost (SSE) and the following graph is achieved. It could be concluded that 'elbow' is formed at K = 5.

x(K)	2	3	4	5	6	7	8	9	10	11
y(SSE)	275249.86	253781.29	231894.76	201356.55	194643.42	174904.78	164926.09	158890.28	151865.86	147402.321

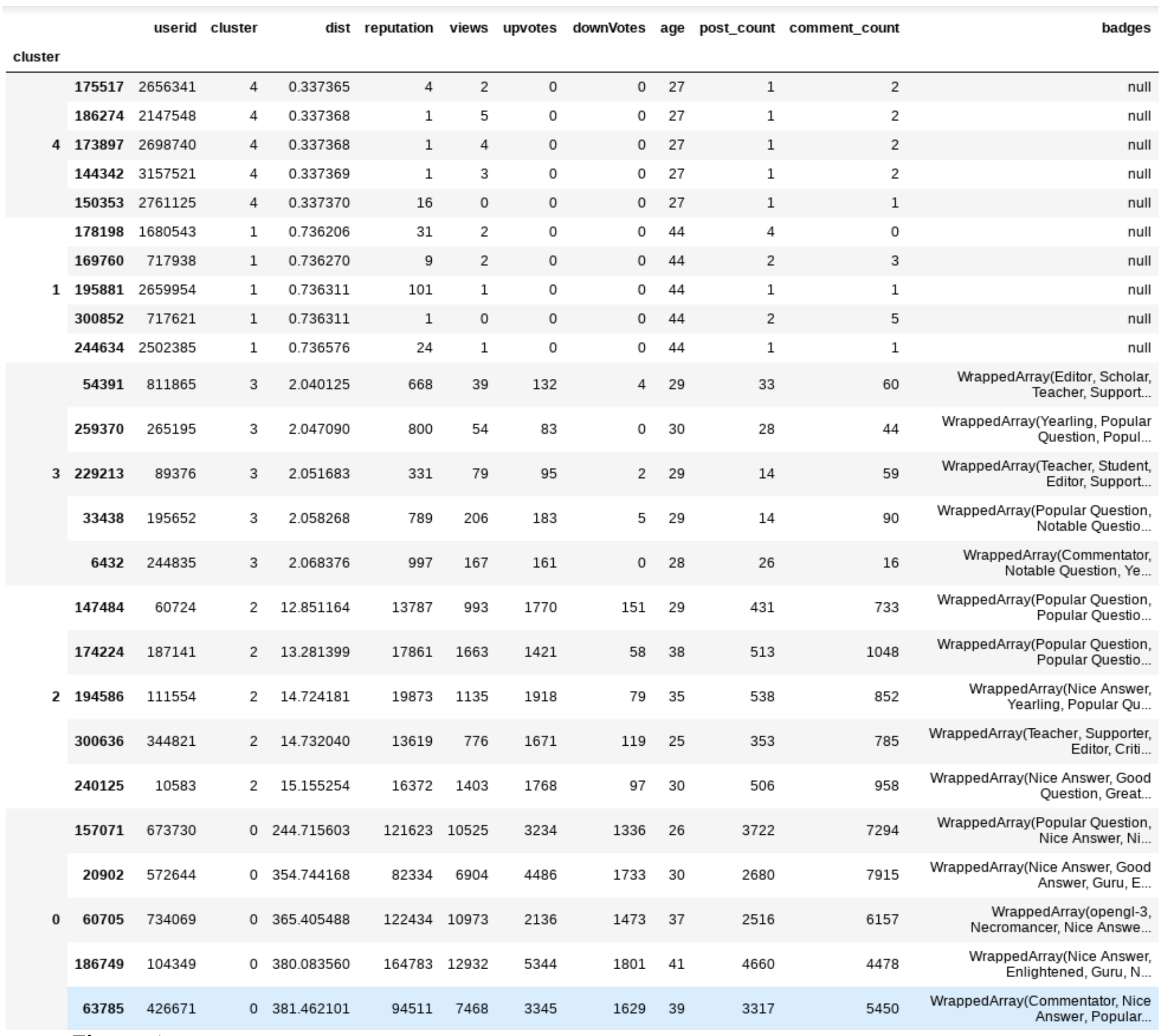


2. Results And Analysis

3.1 Clustering on Users Data

2.1.1 Output

Following is the output obtained after cluster on Users data. Top 5 entries in each cluster sorted in increasing order on the basis of their distance from cluster center, is shown.



cluster		userid	cluster	dist	reputation	views	upvotes	downVotes	age	post_count	comment_count	badges
4	175517	2656341	4	0.337365	4	2	0	0	27	1	2	null
	186274	2147548	4	0.337368	1	5	0	0	27	1	2	null
	173897	2698740	4	0.337368	1	4	0	0	27	1	2	null
	144342	3157521	4	0.337369	1	3	0	0	27	1	2	null
	150353	2761125	4	0.337370	16	0	0	0	27	1	1	null
1	178198	1680543	1	0.736206	31	2	0	0	44	4	0	null
	169760	717938	1	0.736270	9	2	0	0	44	2	3	null
	195881	2659954	1	0.736311	101	1	0	0	44	1	1	null
	300852	717621	1	0.736311	1	0	0	0	44	2	5	null
	244634	2502385	1	0.736576	24	1	0	0	44	1	1	null
3	54391	811865	3	2.040125	668	39	132	4	29	33	60	WrappedArray(Editor, Scholar, Teacher, Support...
	259370	265195	3	2.047090	800	54	83	0	30	28	44	WrappedArray(Yearling, Popular Question, Popul...
	229213	89376	3	2.051683	331	79	95	2	29	14	59	WrappedArray(Teacher, Student, Editor, Support...
	33438	195652	3	2.058268	789	206	183	5	29	14	90	WrappedArray(Popular Question, Notable Questio...
	6432	244835	3	2.068376	997	167	161	0	28	26	16	WrappedArray(Commentator, Notable Question, Ye...
2	147484	60724	2	12.851164	13787	993	1770	151	29	431	733	WrappedArray(Popular Question, Popular Questio...
	174224	187141	2	13.281399	17861	1663	1421	58	38	513	1048	WrappedArray(Popular Question, Popular Questio...
	194586	111554	2	14.724181	19873	1135	1918	79	35	538	852	WrappedArray(Nice Answer, Yearling, Popular Qu...
	300636	344821	2	14.732040	13619	776	1671	119	25	353	785	WrappedArray(Teacher, Supporter, Editor, Criti...
	240125	10583	2	15.155254	16372	1403	1768	97	30	506	958	WrappedArray(Nice Answer, Good Question, Great...
0	157071	673730	0	244.715603	121623	10525	3234	1336	26	3722	7294	WrappedArray(Popular Question, Nice Answer, Ni...
	20902	572644	0	354.744168	82334	6904	4486	1733	30	2680	7915	WrappedArray(Nice Answer, Good Answer, Guru, E...
	60705	734069	0	365.405488	122434	10973	2136	1473	37	2516	6157	WrappedArray(opengl-3, Necromancer, Nice Answe...
	186749	104349	0	380.083560	164783	12932	5344	1801	41	4660	4478	WrappedArray(Nice Answer, Enlightened, Guru, N...
	63785	426671	0	381.462101	94511	7468	3345	1629	39	3317	5450	WrappedArray(Commentator, Nice Answer, Popular...

Figure: 1 **dist = Distance from Center

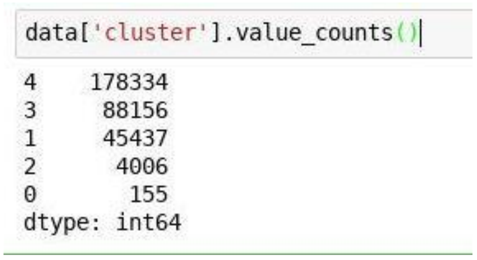
2.1.2 Discussion

Here, we are using human judgement to evaluate the output of the algorithm. The cluster no. 4,1,3,2,0 are in increasing order of user's skillset with cluster no. 0 containing the most skilled Users while 4 containing the least ones.
Following observations can be noted with respect to each feature from Fig1:

- **Reputation:** It increases with the increase in skills in User cluster. It is highest in the cluster '0' and reduces as we approach User cluster with lesser skills like 1 and 4, implying highly skilled users are more reputed on the platform.
- **Views:** Again, it is highest in Cluster'0' and reduces with lesser skilled User cluster, which shows that skilled user have more number of views.
- **UpVotes & DownVotes:** It can be seen that both UpVotes and DownVotes are highest in the highly skilled User cluster '0', which implies that the most skilled users in the database are also the most active users on StackOverflow site. Consequently, both Upvotes & DownVotes are minimum and close to zero in least skilled user cluster '4' indicating their passive behaviour and inactivity.
- **Age:** There is no clear trend when it comes to age, but it can be concluded that most skilled and active users are from late twenties to early forties, owing to their experience, while the least skilled users are mostly in mid-twenties which could be

due to lack of experience at that age.

- **Post_Count & Comment_Count:** As seen in the case of UpVotes and DownVotes, in this case too post_count and comment_count are highest in cluster '0' which represents most skilled users, implying their high activity on the platform. While they are negligible in lesser skilled user cluster '1' & '4' implying idleness.
- **Badges:** It is observed that there are no badges awarded to the unskilled users that lie in the cluster '1' & '4'.
- **Number of Data Points:**



```
data['cluster'].value_counts()

4    178334
3     88156
1     45437
2      4006
0       155
dtype: int64
```

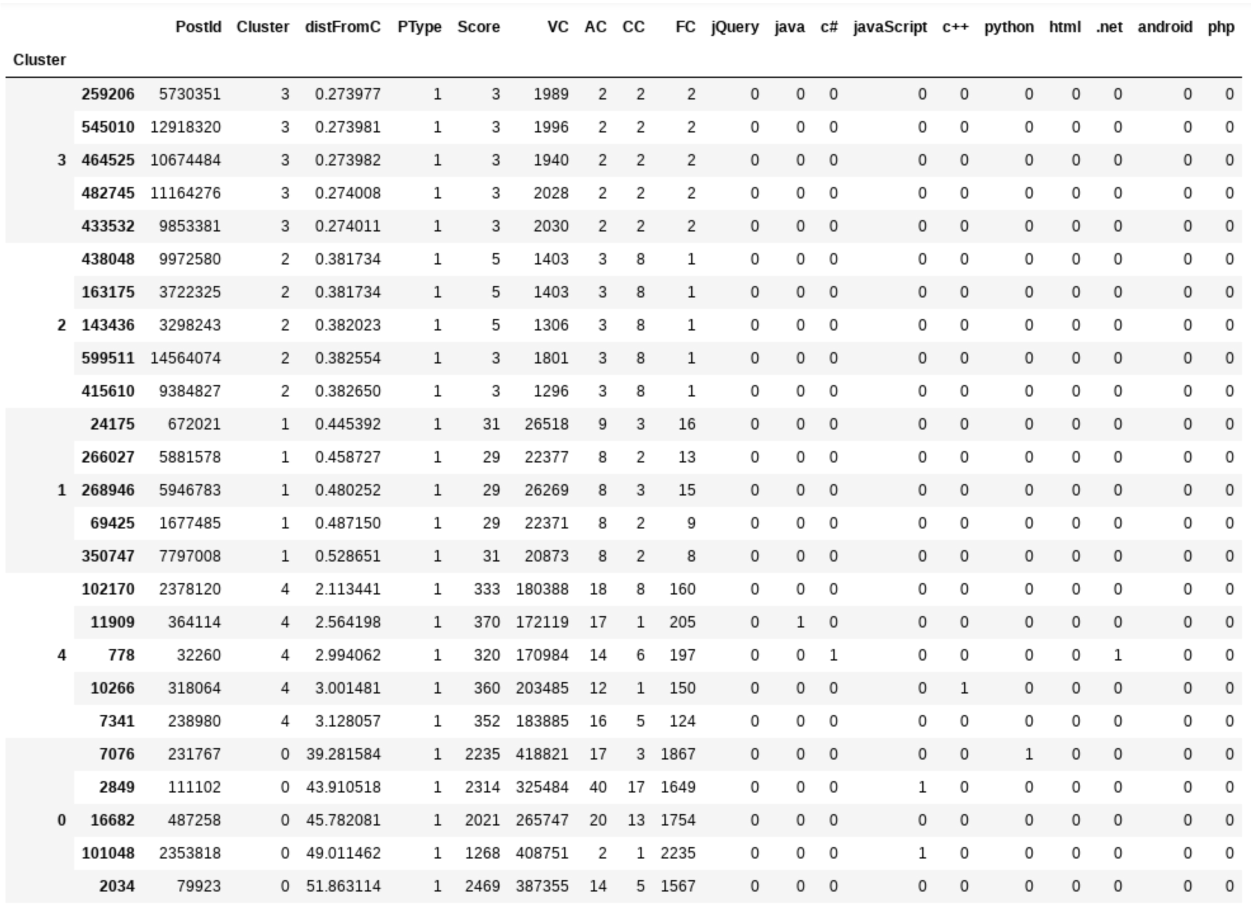
Figure: 2

Fig.2 gives the number of data points in each cluster. Clearly, the cluster with least skilled users is the most populated one while the one containing highly skilled users is the least populated.

2.2 Clustering on Posts Data

2.2.1 Output

Following is the output obtained after cluster on Posts data. Top 5 entries in each cluster sorted in increasing order on the basis of their distance from cluster center, is shown. Also, only 10 most common tags are shown out of 100 due to space constraint.



		PostId	Cluster	distFromC	PType	Score	VC	AC	CC	FC	jQuery	java	c#	javaScript	c++	python	html	.net	android	php
Cluster																				
	259206	5730351	3	0.273977	1	3	1989	2	2	2	0	0	0	0	0	0	0	0	0	0
	545010	12918320	3	0.273981	1	3	1996	2	2	2	0	0	0	0	0	0	0	0	0	0
3	464525	10674484	3	0.273982	1	3	1940	2	2	2	0	0	0	0	0	0	0	0	0	0
	482745	11164276	3	0.274008	1	3	2028	2	2	2	0	0	0	0	0	0	0	0	0	0
	433532	9853381	3	0.274011	1	3	2030	2	2	2	0	0	0	0	0	0	0	0	0	0
	438048	9972580	2	0.381734	1	5	1403	3	8	1	0	0	0	0	0	0	0	0	0	0
	163175	3722325	2	0.381734	1	5	1403	3	8	1	0	0	0	0	0	0	0	0	0	0
2	143436	3298243	2	0.382023	1	5	1306	3	8	1	0	0	0	0	0	0	0	0	0	0
	599511	14564074	2	0.382554	1	3	1801	3	8	1	0	0	0	0	0	0	0	0	0	0
	415610	9384827	2	0.382650	1	3	1296	3	8	1	0	0	0	0	0	0	0	0	0	0
	24175	672021	1	0.445392	1	31	26518	9	3	16	0	0	0	0	0	0	0	0	0	0
	266027	5881578	1	0.458727	1	29	22377	8	2	13	0	0	0	0	0	0	0	0	0	0
1	268946	5946783	1	0.480252	1	29	26269	8	3	15	0	0	0	0	0	0	0	0	0	0
	69425	1677485	1	0.487150	1	29	22371	8	2	9	0	0	0	0	0	0	0	0	0	0
	350747	7797008	1	0.528651	1	31	20873	8	2	8	0	0	0	0	0	0	0	0	0	0
	102170	2378120	4	2.113441	1	333	180388	18	8	160	0	0	0	0	0	0	0	0	0	0
	11909	364114	4	2.564198	1	370	172119	17	1	205	0	1	0	0	0	0	0	0	0	0
4	778	32260	4	2.994062	1	320	170984	14	6	197	0	0	1	0	0	0	0	1	0	0
	10266	318064	4	3.001481	1	360	203485	12	1	150	0	0	0	0	1	0	0	0	0	0
	7341	238980	4	3.128057	1	352	183885	16	5	124	0	0	0	0	0	0	0	0	0	0
	7076	231767	0	39.281584	1	2235	418821	17	3	1867	0	0	0	0	0	1	0	0	0	0
	2849	111102	0	43.910518	1	2314	325484	40	17	1649	0	0	0	1	0	0	0	0	0	0
0	16682	487258	0	45.782081	1	2021	265747	20	13	1754	0	0	0	0	0	0	0	0	0	0
	101048	2353818	0	49.011462	1	1268	408751	2	1	2235	0	0	0	1	0	0	0	0	0	0
	2034	79923	0	51.863114	1	2469	387355	14	5	1567	0	0	0	0	0	0	0	0	0	0

Figure: 3 **Ptype: PostIdType, VC: ViewCount, AC: AnswerCount, CC: CommentCount, FC: FavoriteCount

2.2.2 Discussion

Here again, human judgement is used to evaluate the output of the algorithm. The cluster no. 3,2,1,4,0 are in increasing order of post's popularity with cluster no. 0 containing the most popular posts while 4 containing the least ones.
Following observations can be noted with respect to each feature from Fig3:

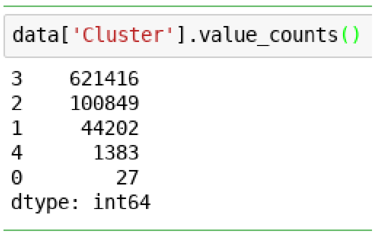
- **Score:** Highly popular post in the cluster 0 have the highest score on the platform while the least popular posts have score close to zero.

- **ViewCount, AnswerCount & FavoriteCount:** All three View count, Answer count and Favorite Count increases with increase in popularity of posts implying high activity on these posts, which also implicitly makes these posts more significant.

- **CommentCount:** There is no explicit trend noticed in this feature. But generally they seem to be larger in cluster '0' with few exceptions.

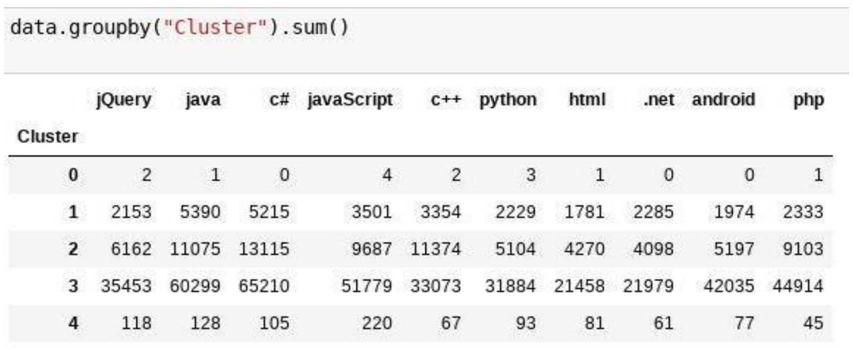
- **Number of Data Points in Cluster:** In fig.4 shown below, it can be seen that the cluster with most popular posts is least populated and on the other hand, cluster with least popular posts is most populated. This implies that there is huge amount of content on the platform which is unattended or unanswered.

- **Tags:** Due to computational constrains it was highly difficult to analyse all tags in the data. But the distribution of 10 most common tags on platform with respect to each cluster are shown in Fig. 5 (below). The results are counter intuitive. Cluster with most popular posts seems to have the least number of most common tags. While in cluster with most unattended and unpopular post, the most common tags are more frequent. One of the reasons could be that Cluster '0' has least number of posts (only 27). It will be interesting to see the tags in cluster '0' (Shown in Fig 6).



```
data['Cluster'].value_counts()

3    621416
2    100849
1     44202
4      1383
0        27
dtype: int64
```

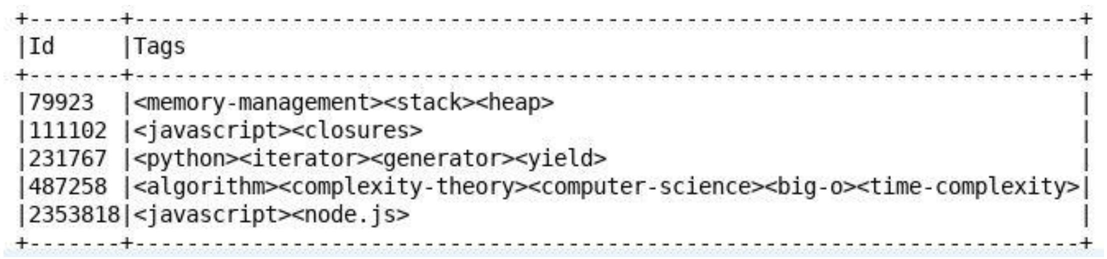
Figure: 4



```
data.groupby("Cluster").sum()
```

Cluster	jQuery	java	c#	javaScript	c++	python	html	.net	android	php
0	2	1	0	4	2	3	1	0	0	1
1	2153	5390	5215	3501	3354	2229	1781	2285	1974	2333
2	6162	11075	13115	9687	11374	5104	4270	4098	5197	9103
3	35453	60299	65210	51779	33073	31884	21458	21979	42035	44914
4	118	128	105	220	67	93	81	61	77	45

Figure: 5



```
+-------+--------------------------------------------------------------------+
|Id     |Tags                                                                |
+-------+--------------------------------------------------------------------+
|79923  |<memory-management><stack><heap>                                    |
|111102 |<javascript><closures>                                              |
|231767 |<python><iterator><generator><yield>                                |
|487258 |<algorithm><complexity-theory><computer-science><big-o><time-complexity>|
|2353818|<javascript><node.js>                                               |
+-------+--------------------------------------------------------------------+
```

Figure: 6

3. Implementation using Mlib Package

Results are also obtained after implementing Kmeans with the help of Mlib Package.

```
clusters = KMeans.train(datardd, 5,maxIterations =100,initializationMode = "random")
```

Here, is the comparison table,

K =5	SSE using Mlib	SSE without using Mlib
Users Clustering	536013.630268	554363.320165
Posts Clustering	1014635.68393	1032492.738614

III CONCLUSION AND FUTURE WORK

As seen from above table, SSE obtained in the case where Mlib is used is slightly lesser in comparison. It can be concluded that the simplicity and scalability of the package helps to train model faster leading to lower learning curves and better results.

In this project, Kmeans algorithm was applied to group similar Users and Posts with respect to different features and evaluated the results on the basis of human judgement. A quantitative evaluation metric could be also used for evaluation like Silhoutte Scores, Rand index or Confusion Matrix. Various other clustering algorithms could be also applied and compared their results with K-means. Lastly, extra features like Time and Date can be also added to analyse posts or users more efficiently.