

# Homework Assignment 1(Programming Category)

Student Name: Vijay Saravana Jaishanker

Student Session: cs6220-A

I have chosen to do **Option 1.2** for HW1.

## Deliverable 1:

Source code and executable with readme. (provide URL if you use open source code packages)

I have attached the source code, executables and README in the zip file. You can also access them in my github link for HW1.

**Github Link for HW1** : <https://github.gatech.edu/vjaishanker3/Vijay-CS6220/tree/main/HW1>

## References :

Hadoop Word count : [Link](#)

Spark Word count : <https://pythonexamples.org/pyspark-word-count-example/>


Spark K Means : <https://www.data4v.com/tutorial-k-means-clustering-on-spark/>

## Deliverable 2:

Screen Shots of your execution process/environments:

I have attached the screenshots showing the running of the Hadoop MapReduce, Apache Spark and also the Spark ML Lib execution in the Jupyter notebook. I have attached all screenshots in the **`screenshots`** folder of every part.

Sample Screenshots:



Application application\_1662610626524\_0009

Logged in as: dr.who

Cluster

About
Nodes
Node Labels
Applications
NEW
NEW SAVING
SUBMITTED
ACCEPTED
RUNNING
FINISHED
FAILED
KILLED
Scheduler
Tools

Application Overview

User: ubuntu
Name: word count
Application Type: MAPREDUCE
Application Tags:
Application Priority: 0 (Higher Integer value indicates higher priority)
YarnApplicationState: FINISHED
Queue: default
FinalStatus Reported by AM: SUCCEEDED
Started: Thu Sep 08 01:40:26 -0400 2022
Launched: Thu Sep 08 01:40:26 -0400 2022
Finished: Thu Sep 08 01:40:42 -0400 2022
Elapsed: 15sec
Tracking URL: History
Log Aggregation Status: DISABLED
Application Timeout (Remaining Time): Unlimited
Diagnostics:
Unmanaged Application: false
Application Node Label expression: <Not set>
AM container Node Label expression: <DEFAULT\_PARTITION>

Application Metrics

Total Resource Preempted: <memory:0, vCores:0>
Total Number of Non-AM Containers Preempted: 0
Total Number of AM Containers Preempted: 0
Resource Preempted from Current Attempt: <memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt: 0
Aggregate Resource Allocation: 52146 MB-seconds, 27 vcore-seconds
Aggregate Preempted Resource Allocation: 0 MB-seconds, 0 vcore-seconds

Show: 20 entries

Search:

Attempt ID	Started	Node	Logs	Nodes blacklisted by the app	Nodes blacklisted by the system
appattempt_1662610626524_0009_000001	Thu Sep 8 01:40:26 -0400 2022	http://MSI.localdomain:8042	Logs	0	0

Showing 1 to 1 of 1 entries

First
Previous
1
Next
Last

Fig 1: Hadoop Map reduce - Word count program execution.

## Find optimal number of clusters using the silhouette method

```
In [4]: # Array to hold silhouette scored for all values of K
silhouette_scores=[]

evaluator = ClusteringEvaluator(featuresCol='iris_features', \
metricName='silhouette', distanceMeasure='squaredEuclidean')

for K in range(2,11):

    KMeans_ = KMeans(featuresCol='iris_features', k=K)

    # Fit the Iris dataset onto KMeans model
    KMeans_fit=KMeans_.fit(assembled_data)

    KMeans_transform=KMeans_fit.transform(assembled_data)

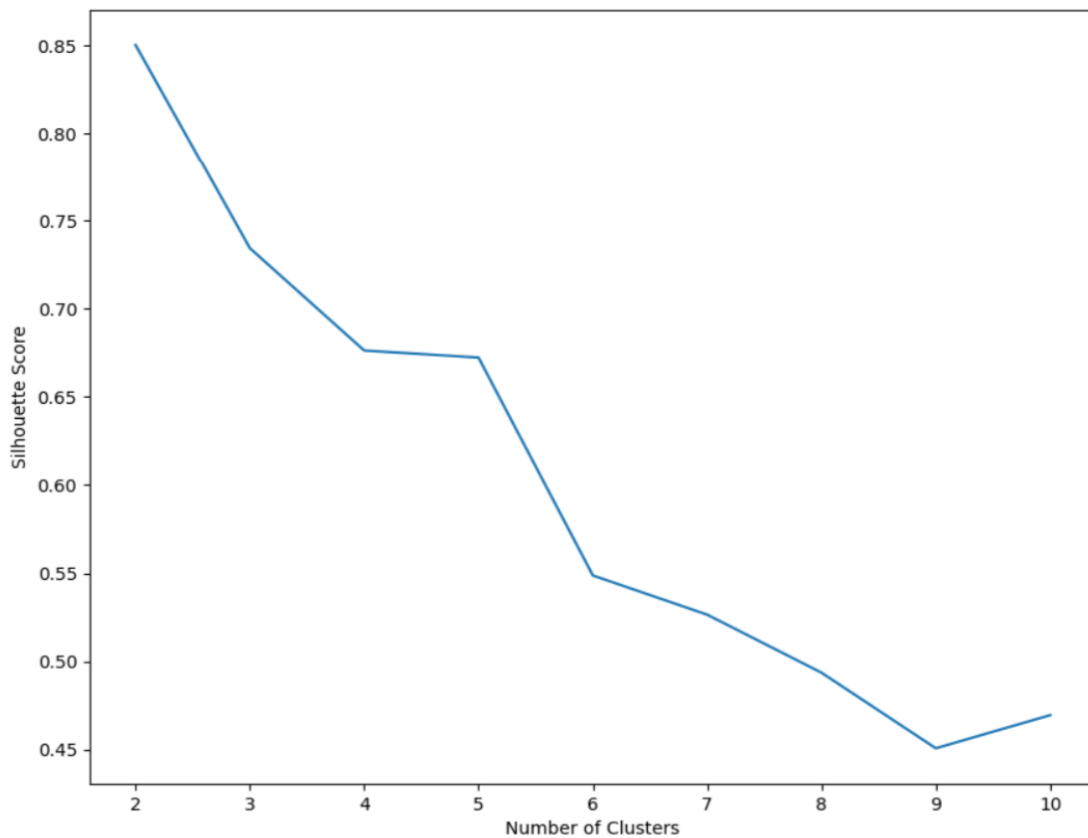
    evaluation_score=evaluator.evaluate(KMeans_transform)

    silhouette_scores.append(evaluation_score)
```

## Plot the silhouette score vs number of clusters (Elbow Method)

```
In [5]: fig, ax = plt.subplots(1,1, figsize =(10,8))
ax.plot(range(2,11),silhouette_scores)
ax.set_xlabel('Number of Clusters')
ax.set_ylabel('Silhouette Score')
```

```
Out[5]: Text(0, 0.5, 'Silhouette Score')
```



Using elbow method, we can see that the first elbow occurs at K = 3, showing that 3 clusters is the optimal value.

Fig 2: Spark ML Lib - KMeans clustering execution in Jupyter Notebook

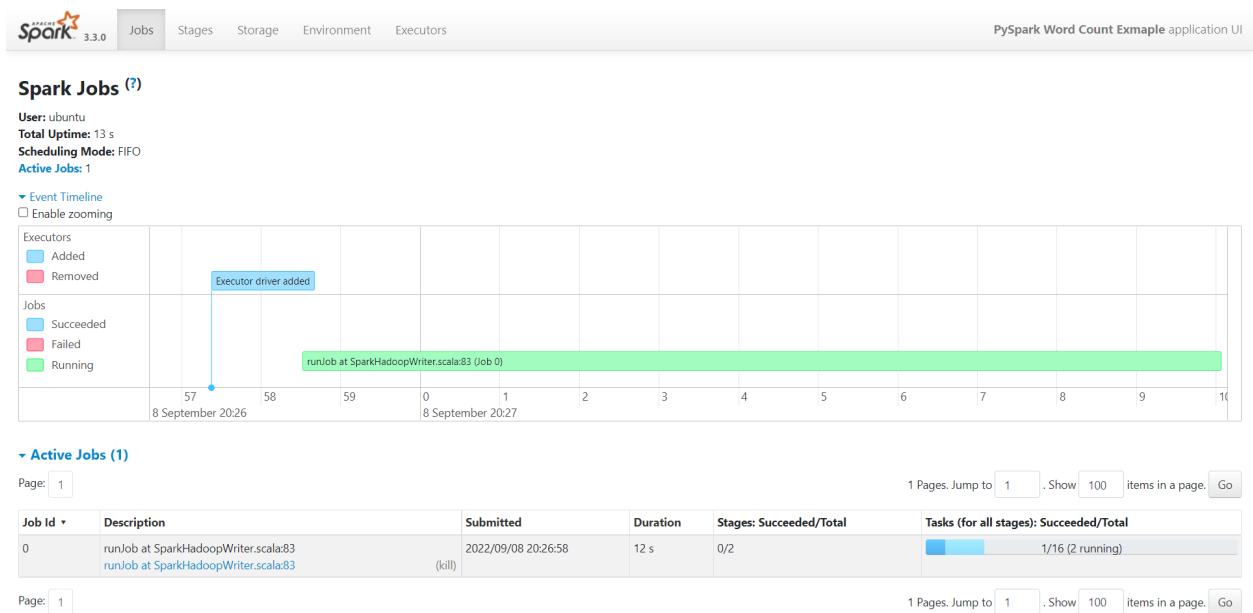


Fig 3: Example showing spark execution for Word count program

### Deliverable 3:

Input and output of your program

I have mentioned in detail about the inputs and outputs for each program in the readme file for each part (Hadoop, Spark and KMeans). I am again mentioning it here.

#### Hadoop - word count program:

##### Inputs:

We used 8 files of varying sizes to run the wordcount program - the files are of sizes as mentioned below :

Filename	File size (kB)
File1.txt	1680
File2.txt	3472
File3.txt	5178
File4.txt	6463
File5.txt	9766
File6.txt	97657
File7.txt	976563
File8.txt	9765625

File1 to File4 are samples of the text file at <https://norvig.com/big.txt>. File5 to File8 are autogenerated large text files using the following command by incrementing the file size by order of 10 every file:

```
$ base64 /dev/urandom | head -c 10000000 > File5.txt
```

### **Outputs:**

The outputs of first 4 files are placed in the outputs folder. The remaining are too big to store on github. The output files have the word and the number of occurrences present in the following fashion:

```
hello 2  
bye 3
```

The screenshots folder has the screenshots of the execution the in Apache Hadoop GUI when it is running on the local cluster.

### **Spark - word count program:**

#### **Inputs:**

We used 8 files of varying sizes to run the wordcount program - the files are of sizes as mentioned below :

Filename	File size (kB)
File1.txt	1680
File2.txt	3472
File3.txt	5178
File4.txt	6463
File5.txt	9766
File6.txt	97657
File7.txt	976563
File8.txt	9765625

File1 to File4 are samples of the text file at <https://norvig.com/big.txt>. File5 to File8 are autogenerated large text files using the following command by incrementing the file size by order of 10 every file:

```
$ base64 /dev/urandom | head -c 10000000 > File5.txt
```

## Outputs:

The outputs of 2 files (File4 and File5) are placed in the outputs folder. The remaining are too big to store on github. The output files have the word and the number of occurrences present in the following fashion:

(hello, 2)  
(bye, 3)

The screenshots folder has the screenshots of the execution the in Spark GUI when it is running on the local cluster. It contains details of the Spark executor, spark job stages and so on.

## Spark - KMeans clustering program:

### Inputs:

The input to the program is the Iris Dataset. <https://archive.ics.uci.edu/ml/datasets/iris>

Excerpts from the dataset description:

--- The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2.  
--- Predicted attribute: class of iris plant.

### Outputs:

We have performed the Clustering of the classes in the dataset by using the elbow method and also visually representing the different clusters using seaborn library.

The notebook has all the details and outputs from the code run.

## Deliverable 4:

Runtime measurements in excel plots or tabular format.

The results have been recorded and analysed in the [excel sheet here](#). Below are the various results for running the wordcount program using Apache Hadoop and Apache Spark respectively.

Filename	File size (kB)	Elapsed Time (sec)
File1.txt	1680	15
File2.txt	3472	15

File3.txt	5178	16
File4.txt	6463	16
File5.txt	9766	13
File6.txt	97657	17
File7.txt	976563	152
File8.txt	9765625	483

Table 1: Apache Hadoop - Word count program runtime for various file sizes.

### Hadoop Map Reduce - Word Count

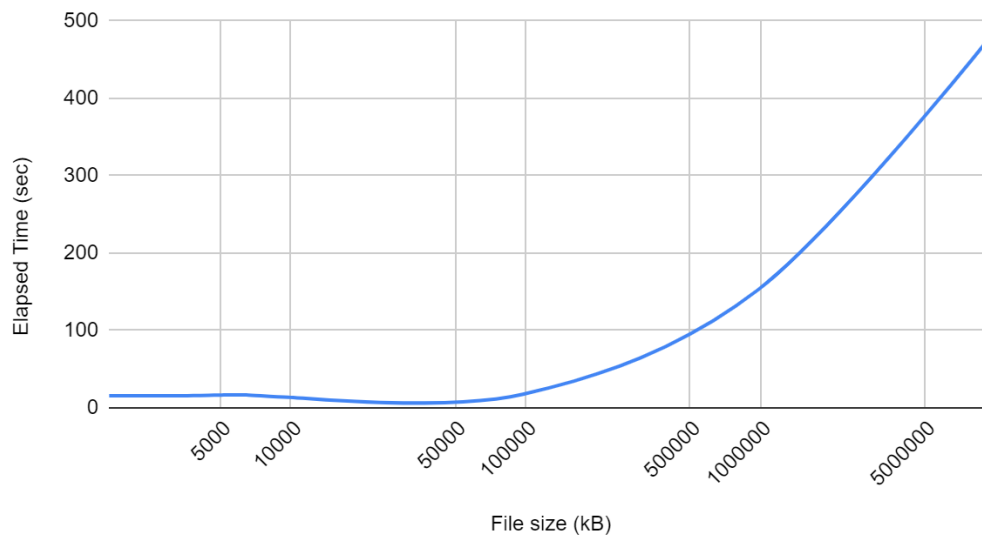


Chart 1: Apache Hadoop - Word count program runtime for various file sizes (Log).

Filename	File size (kB)	Elapsed Time (sec)
File1.txt	1680	1.6
File2.txt	3472	1.6
File3.txt	5178	1.5
File4.txt	6463	1.7
File5.txt	9766	1.4
File6.txt	97657	6.3
File7.txt	976563	61.3
File8.txt	9765625	693.6

Table 2: Apache Spark - Word count program runtime for various file sizes.

### Spark - Word count program

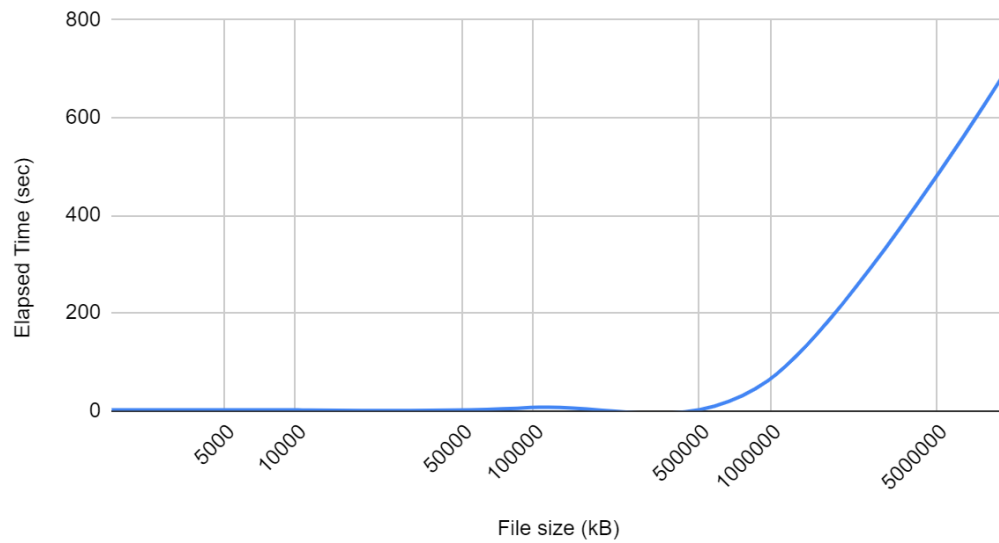


Chart 2: Apache Spark- Word count program runtime for various file sizes (Log).

### Spark vs Hadoop - Word count performance

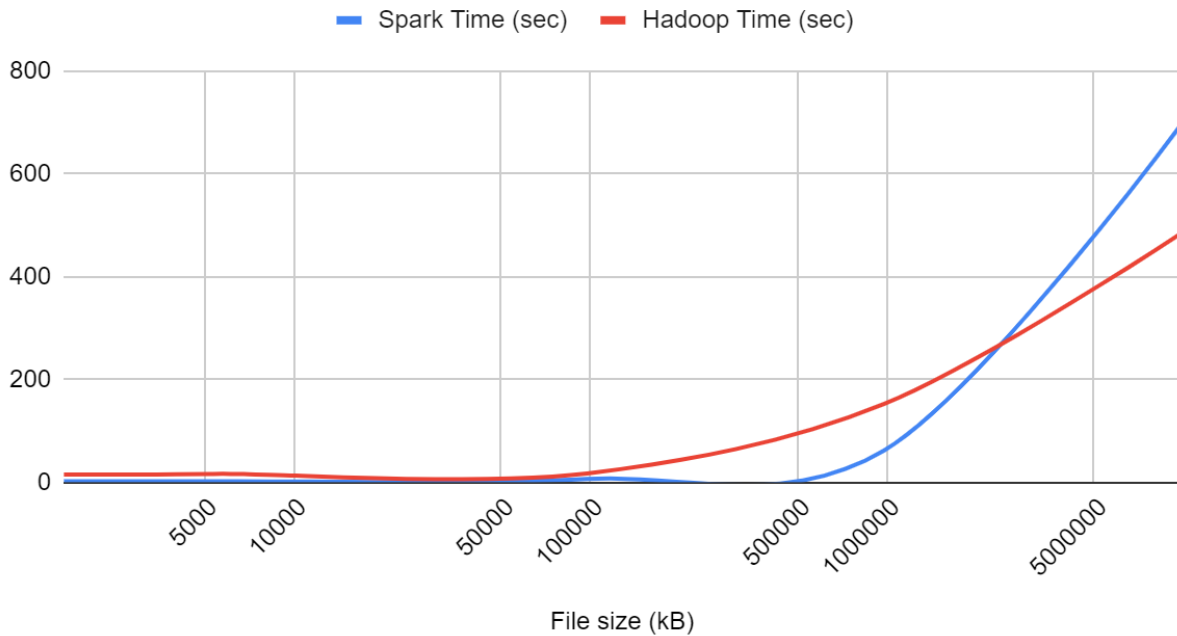


Chart 3: Apache Spark vs Apache Hadoop - Word count program runtime comparison (Log).



## **Deliverable 5:**

Two scenarios that you choose to run the same MapReduce program to compare the performance or two programs solving the same problem with the same dataset.

We have run the same word count program over input files of different file sizes starting from 1000kb all the way upto 10GB and we have tabulated the results in the table mentioned above. We have performed this experiment for both Apache spark and apache hadoop.

## **Deliverable 6:**

Provide your analytical report.

It can be seen that there is no visible increase in the time taken to execute from 1000kB to 10000kB. Performance is almost similar. The constant time could be due to the cluster spin up and tear down overhead. From the logs, we can also see that the system uses just 1 split - that is the data partitioning is not required since the amount of data can be handled with a single partition. The parallelism is rather limited in this case as we can see the system is fast enough to handle the data size.

However, as we increase the data size from 10000 kB to 100000 kB we can see that the time taken to execute increases 10x for both hadoop and spark. Increasing the data size further from 100000 kB to 1000000 kB - increases the runtime 5x in Hadoop and 10x in Spark.

Spark performs better than hadoop till 100000 kB file size - this could be attributed to in-memory computations of spark as compared to the time required to write to disk for Hadoop. Spark also has other improvements in comparison to Hadoop that it supports the concept of Data Lineage using RDD (Resilient Distributed Datasets). Instead of performing the action on the data, spark will do a lazy evaluation and record the transformations and perform computation when a show() operation is performed. This leads to additional possibilities for optimization.

However, hadoop performs much better for large file size (10 GB) as compared to spark - this is again due to spark being unable to accommodate all the RDD partitions in memory, so it has to write to disk which leads to poor performance. We can also observe that till File 6, the number of partitions created of the file is 1, but for File 7, the system creates 8 splits/ partitions. For File 8, the system creates 75 partitions showing how it manages large data workloads by partitioning and parallel processing of data.

From the charts 1 to 3, we can see the performance of spark vs Hadoop and how the performance varies with increasing file sizes. This explains the case study shown in class as shown in the figure below:

Configuration of host machines: 96 GB DRAM, 2.67GHz Intel Xeon, 8 cores w. hyperthreading.  
Each runs 8 Spark executors, 10 GB Heap, 0.6 memory fraction for RDD caching.

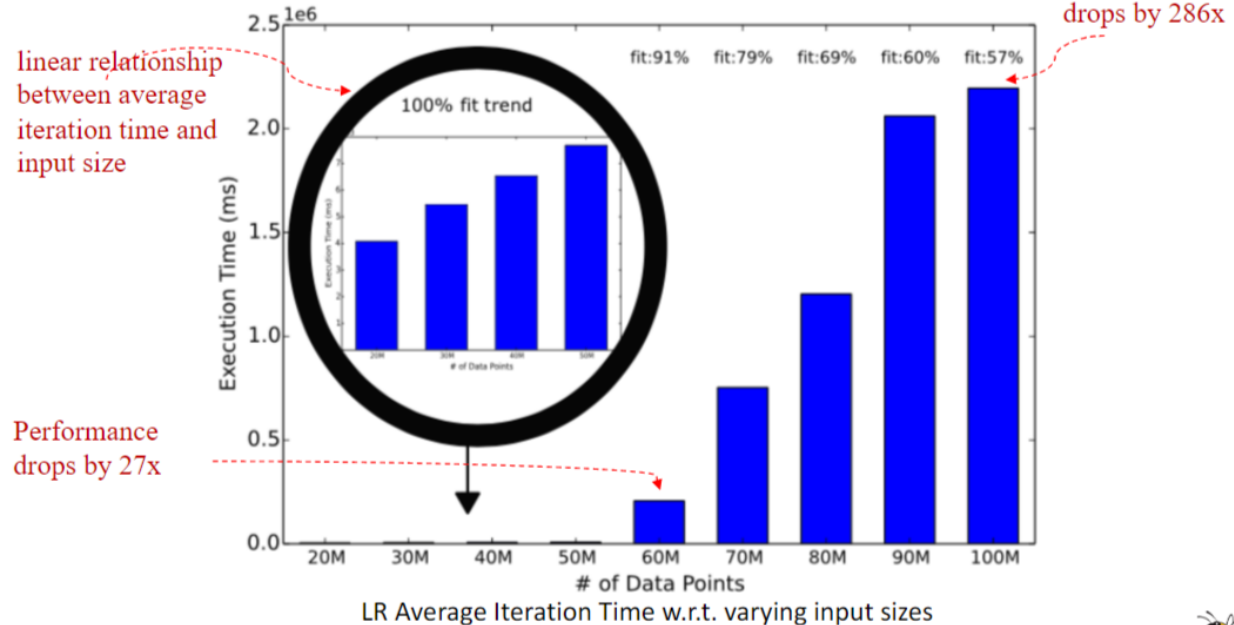


Fig 4 : Performance drop in spark with increasing data quantity

### KMeans using Spark ML Lib:

KMeans clustering is an algorithm which is used to separate the data points into classes based on the number of clusters. Calculating a cluster involves choosing a center point and calculating the euclidean distance for all the points from that point. This could be a computation heavy operation considering the number of euclidean distance calculations we need to do depending on the number of datapoints.

Spark ML Lib has the ability to parallelize this computation heavy operation and it is abstracted under the ML Lib library. The source code is present in the jupyter notebook and has the code and outputs for number of cluster calculation using Elbow Method (Silhouette) and then fitting the datapoint on the model with K clusters which is calculated before. Spark uses the concept of Data parallelism and divides the dataset into multiple RDD partitions and divides them among spark executor cores each of which processes the partition and performs the spark job (which is to calculate the euclidean distances).

We use the Iris dataset as the input to perform the clustering and to reduce the 4D data to show the clustering on a 2D plane, we perform PCA (Principal component Analysis) to show only the top 2 components and the clusters as below.

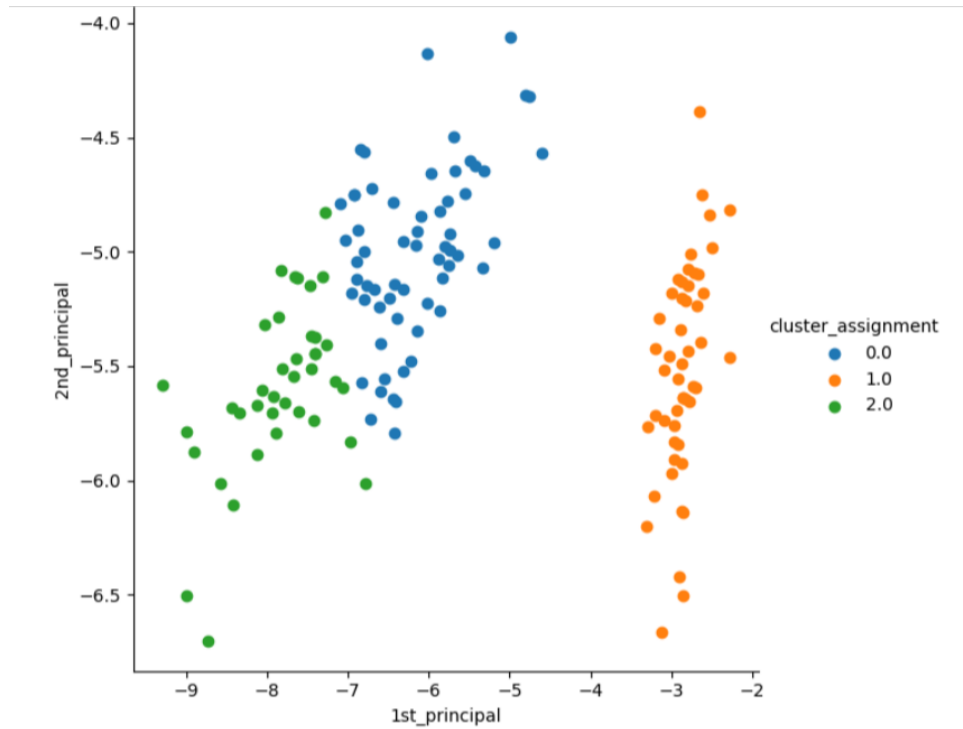


Fig 5: Clustering using KMeans in Spark ML Lib

Thus, we have shown how K Means clustering can be performed in a distributed manner efficiently using Spark ML Lib. Please refer to the [github link](#) for more details and information.