تمرین اول درس Big Data

عرفان اشتری 401123916

1)

الف)

در ابتدا کتابخانه های مورد نیاز را فرا میخوانیم و یک فضای pyspark ایجاد میکنیم.

from pyspark.sql import SparkSession #Import the pyspark  
from pyspark.conf import SparkConf #Import the SparkConf  
from pyspark.context import SparkContext #Import the SparkContext  
import time

#initializing and starting a pyspark session  
  
conf = SparkConf()  
conf.setMaster("local").setAppName("word-counts")   
sc = SparkContext.getOrCreate(conf=conf)

در ادامه در شروع کار باید ابتدا فایل متنی مورد نظر را توسط اسپارک بخوانیم و سپس تعداد کل کلمات را بخوانیم، جداسازی کلمات را با space (فاصله = " ") در نظر میگیریم و با استفاده از کد زیر یک لیست به دست می آید که هر عضو آن یک تاپل میباشد که شامل کلمه و تعداد تکرار آن است که با جمع کردن همه اعداد به تعداد کل کلمات میرسیم.

#reading all the words in the text file and making a list of tupples including words and their counts  
lines = sc.textFile("JaneAusten.txt") #The file is read using the textFile() method.  
word\_counts = lines.flatMap(lambda line: line.split(' '))\  
 .map(lambda word: (word,1)) \  
 .reduceByKey(lambda count1, count2: count1 + count2) \  
 .collect()   
  
  
c = 0  
  
#counting all the words  
for (word,count) in word\_counts[0:]:  
# print(word,count)  
 c += count  
   
print("count = ",c)  
  
c = 0  
  
#counting words except for space(" ")   
for (word,count) in word\_counts[1:]:  
# print(word,count)  
 c += count  
   
print("count words except for space = ",c)

در این کد یکبار تعداد کل کلمات بدست آمده و یکبار هم تعداد بدون در نظر گرفتن space و مقادیر آن ها به ترتیب 797474 و 780223 میباشد.

لیست word\_counts به شکل زیر می باشد.

[('', 17251),  
 ('Project', 81),  
 ("Gutenberg's", 1),  
 ('The', 1663),  
 ('Complete', 3),  
 ('Works', 3),  
 ('of', 22761),  
 ('Jane', 374),  
 ('Austen,', 6),  
 .

.

.  
 ('utility', 6),  
 ('abroad,', 11),  
 ('talents', 16),  
 ('accomplishments', 9),  
 ...]

ب)

در این بخش میخواهیم تعداد کلمات بدون تکرار را بدست آورریم برای این کار پس از خواندن فایل متنی مطابق بخش قبل، از کد و توابع زیر استفاده میکنیم.

distinct\_words\_counts = words\_counts.distinct() #The unique words can be found by invoking the distinct() function on the rdd.  
  
  
count = distinct\_words\_counts.count() #The count of unique words is obtained by invoking the count() function.

با استفاده از این کدها مقدار count که همان تعداد لغات بدون تکرار است برابر با 44360 بدست آمد.

ج)

در این بخش ابتدا از لیست word\_counts استفاده میکنیم و آن را برحسب تعداد تکرار کلمات مرتب میکنیم.

#sorting files based on their counts and showing top 10  
sort = sorted(word\_counts,key=lambda l:l[1], reverse=True)  
sort[:10]

در نهایت 10 کلمه ای که بیشترین تکرار دارند کلمات زیر میباشند.

[('the', 26654),  
 ('to', 25322),  
 ('of', 22761),  
 ('and', 22184),  
 ('', 17251),  
 ('a', 13772),  
 ('I', 11686),  
 ('in', 11523),  
 ('her', 11427),  
 ('was', 11342)]

در ادامه برای قسمت بعدی سوال باید شمارش تعداد کلمات بدون تکرار را با استفاده از تعداد 1 تا 4 هسته از پردازشگر انجام دهیم که برای تغییر تعداد هسته ها میبایست در تنظیمات مربوط به راه اندازی اسپارک conf.setMaster(f"local[{i}]").setAppName("word-counts") قرار دهیم که در آن i نشان دهنده تعداد هسته هاست. کد این بخش به صورت زیر میباشد.

#changing the number of processors for exprimenting the process time  
T = []  
num\_cores= [ 1,2,3,4]  
for i in num\_cores:  
 conf = SparkConf()  
 conf.setMaster(f"local[{i}]").setAppName("word-counts") # set the number of cores  
 sc = SparkContext.getOrCreate(conf=conf) #The spark context object is assigned to a variable sc.  
 print(conf.getAll())  
  
 t = time.time()  
  
 lines = sc.textFile("JaneAusten.txt") #The file is read using the textFile() method.  
  
 words\_rdd = lines.flatMap(lambda line: line.split(' '))  
  
 distinct\_words\_counts = words\_rdd.distinct() #The unique words can be found by invoking the distinct() function on the rdd.  
  
  
 count = distinct\_words\_counts.count() #The count of unique words is obtained by invoking the count() function.  
  
 print("core:",i)  
   
 print(time.time() - t)  
 T.append(time.time() - t)  
  
 print("The count of unique words in the file is:", count)

در این بخش لیست num\_cores شامل تعداد هسته استفاده شده و لیست T شامل زمان انجام عملیات برای هر تعداد هسته است. برای به دست آوردن زمان از time.time() در ابتدا و انتهای عملیات مورد نظر استفاده شده است. در پایان خروجی زیر برای این بخش بدست آمد

[('spark.app.name', 'word-counts'), ('spark.submit.pyFiles', ''), ('spark.submit.deployMode', 'client'), ('spark.app.submitTime', '1672598316727'), ('spark.ui.showConsoleProgress', 'true'), ('spark.master', 'local[1]')]  
core: 1  
2.042257070541382  
The count of unique words in the file is: 44360

[('spark.app.name', 'word-counts'), ('spark.submit.pyFiles', ''), ('spark.submit.deployMode', 'client'), ('spark.app.submitTime', '1672598316727'), ('spark.ui.showConsoleProgress', 'true'), ('spark.master', 'local[2]')]  
core: 2  
1.5026111602783203  
The count of unique words in the file is: 44360

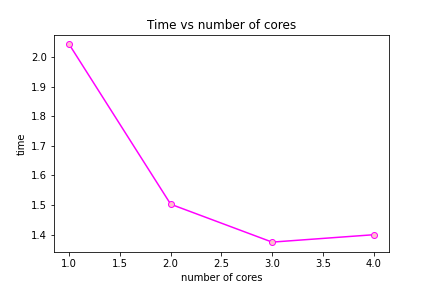
[('spark.master', 'local[3]'), ('spark.app.name', 'word-counts'), ('spark.submit.pyFiles', ''), ('spark.submit.deployMode', 'client'), ('spark.app.submitTime', '1672598316727'), ('spark.ui.showConsoleProgress', 'true')]  
core: 3  
1.3745746612548828  
The count of unique words in the file is: 44360

[('spark.master', 'local[4]'), ('spark.app.name', 'word-counts'), ('spark.submit.pyFiles', ''), ('spark.submit.deployMode', 'client'), ('spark.app.submitTime', '1672598316727'), ('spark.ui.showConsoleProgress', 'true')]  
  
[Stage 590:> (0 + 1) / 1]  
core: 4  
1.3995206356048584  
The count of unique words in the file is: 44360

و با استفاده از کد زیر نمودار مربوط به آن رسم گردید.

import matplotlib.pyplot as plt  
  
plt.plot(num\_cores,T, color='magenta', marker='o',mfc='pink' ) #plot the data  
  
plt.ylabel('time') #set the label for y axis  
plt.xlabel('number of cores') #set the label for x-axis  
plt.title("Time vs number of cores") #set the title of the graph  
plt.show() #display the graph  
plt.savefig("Time vs number of cores.png")

که نتیجه آن به صورت زیر است.



همانطور که قابل مشاهده است با افزایش تعداد هسته از یک به چهار زمان پردازش که برحسب ثانیه است کاهش یافته و تقریبا ثابت شده است و در واقع سرعت پردازش با تعداد هسته های بیشتر افزایش می یابد.

2)

الف)

در این سوال همانند سوال قبل سه فایل متنی مورد نظر را میخوانیم، برای این کار ابتدا کتابخانه های موردنیاز فراخوانده شده و پس از راه اندازی pyspark به سراغ خواندن فایل ها می رویم. کد مورد نیاز برای خواندن فایل متنی اول برای مثال در زیر آمده است.

import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
from mpl\_toolkits.mplot3d import Axes3D  
from pyspark import SparkContext  
from pyspark.ml.clustering import KMeans, BisectingKMeans  
from pyspark.ml.feature import VectorAssembler  
from pyspark.sql import SQLContext  
from pyspark.sql import SparkSession #Import the pyspark  
from pyspark.conf import SparkConf #Import the SparkConf  
from pyspark.context import SparkContext #Import the SparkContext  
import time

#initializing and starting a pyspark session  
conf = SparkConf()  
conf.setMaster("local").setAppName("word-counts")   
  
sc = SparkContext.getOrCreate(conf=conf)  
  
df1 = sc.textFile("C1.txt") #The file is read using the textFile() method.

پس از خواندن فایل متنی و ذخیره آن در دیتافریم اسپارک df1 باید ابتدا اطلاعات عددی موجود در آن ها که درواقع همان مختصات میباشند استخراج شود و در نهایت در یک دیتافریم و فایل csv ذخیره شود تا در ادامه راحت تر بتوان از آن استفاده کرد.

# save the numbers in columns of dataframe as lists  
L1 = []  
L2 = []  
for i in range(df1.count()):  
 m= df1.collect()[i].find("\t")  
   
 L1.append(df1.collect()[i][:m])  
 L2.append(df1.collect()[i][m+1:])  
   
# making a pandas dataframe using the lists  
C1\_list = pd.DataFrame(  
 {'X': L1,  
 'Y': L2  
 })  
  
# save the dataframe as a csv file  
C1\_list.to\_csv('C1\_pd.csv', index=False)

سپس دومرتبه دیتا را با استفاده از کد زیر میخوانیم.

sqlContext = SQLContext(sc)

FEATURES\_COL = ['X', 'Y']  
path = 'C1\_pd.csv'

# reading csv file as pyspark dataframe  
df = sqlContext.read.csv(path, header=True)   
df.show()

که نتیجه به صورت زیر است

+------+------+  
| X| Y|  
+------+------+  
|624474|837604|  
|673412|735362|  
|647442|677000|  
|532283|741384|  
|646529|742844|  
|647535|755101|  
|644131|777721|  
|521368|736923|  
|688940|798967|  
|592666|805244|  
|645068|716248|  
|666740|707391|  
|662064|644958|  
|630628|689662|  
|623268|774834|  
|617896|766560|  
|560260|629298|  
|595728|703618|  
|594177|665266|  
|544862|802997|  
+------+------+  
only showing top 20 rows

در ادامه چون ما نیاز داریم جنس دیتا عددی باشد و نه رشته، مقدارهای دیتافریم را به نوع float تغییر میدهیم.

df\_feat = df.select(\*(df[c].cast("float").alias(c) for c in df.columns[:]))

در نهایت هر دو ستون را به یک ستون واحد به نام features تبدیل میکنیم که هر سطر آن شامل یک لیست میباشد که دو مختصات را در خود دارد. کد این بخش به صورت زیر است.

#adding a features column which contains all the columns as a single list  
  
vecAssembler = VectorAssembler(inputCols=FEATURES\_COL, outputCol="features")  
# choosing just the features column for a new dataframe  
df\_kmeans = vecAssembler.transform(df\_feat).select('features')  
df\_kmeans.show()

حال در df\_kmeans دیتا فریمی داریم که فقط یک ستون features دارد که همه اطلاعات مورد نیاز را دارد.

+-------------------+  
| features|  
+-------------------+  
|[624474.0,837604.0]|  
|[673412.0,735362.0]|  
|[647442.0,677000.0]|  
|[532283.0,741384.0]|  
|[646529.0,742844.0]|  
|[647535.0,755101.0]|  
|[644131.0,777721.0]|  
|[521368.0,736923.0]|  
|[688940.0,798967.0]|  
|[592666.0,805244.0]|  
|[645068.0,716248.0]|  
|[666740.0,707391.0]|  
|[662064.0,644958.0]|  
|[630628.0,689662.0]|  
|[623268.0,774834.0]|  
|[617896.0,766560.0]|  
|[560260.0,629298.0]|  
|[595728.0,703618.0]|  
|[594177.0,665266.0]|  
|[544862.0,802997.0]|  
+-------------------+  
only showing top 20 rows

همین کارها را برای دو فایل متنی دیگر نیز انجام می دهیم که اطلاعتشان در نوت بوک موجود است. حالا که دیتا فریم آماده شده است به بخش بعدی میرسیم که اعمال روش های خوشه بندی میباشد، در ابتدا روش خوشه بندی kmeans++ را اعمال میکنیم که برای این کار باید از تابع خوشه بندی kmeans اسپارک استفاده کنیم و مقدار پارامتر initMode را برابر با 'k-means||' قرار دهیم و برای تنظیم تعداد کلاستر از .set(k) استفاده شده که k نشان دهنده تعداد کلاستر است که از 2 تا 25 متغیر است. پس از پایان عملیات برای هر کلاستر مقدار هزینه آن از روش Silhouette score را در لیست cost ذخیره میکنیم که در ادامه برای رسم نمودار استفاده خواهد شد. کد این بخش برای kmeans++ به صورت زیر است.

# training a kmeans ++ (initMode='k-means||') model on the dataframe extracted from the text file  
# evaluating using Silhouette methode  
from pyspark.ml.evaluation import ClusteringEvaluator  
cost = np.zeros(25)  
for k in range(2,25):  
 kmeans = KMeans(initMode='k-means||').setK(k).setSeed(1).setFeaturesCol("features")  
 model = kmeans.fit(df\_kmeans)  
  
 pdt = model.transform(df\_kmeans)  
 evaluator = ClusteringEvaluator()  
 cost[k] = evaluator.evaluate(pdt) #Silhouette with squared euclidean distance

و برای خوشه بندی با Bisecting K-means هم از کد زیر استفاده میکنیم.

# training a BisectingKMeans model on the dataframe extracted from the text file  
# evaluating using Silhouette methode  
from pyspark.ml.evaluation import ClusteringEvaluator  
cost = np.zeros(25)  
for k in range(2,25):  
 bkm = BisectingKMeans().setK(k).setSeed(1).setFeaturesCol("features")  
 model = bkm.fit(df\_kmeans)  
  
 pdt = model.transform(df\_kmeans)  
 evaluator = ClusteringEvaluator()  
 cost[k] = evaluator.evaluate(pdt) #Silhouette with squared euclidean distance

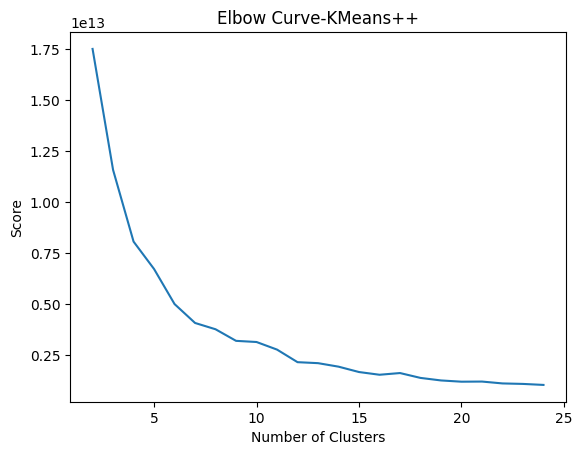
با اعمال این کدها روی دیتافریم های بدست آمده از فایل های متنی به مقدار هزینه یا همان Silhouette score برای هر تعداد کلاستر بین 2 تا 25 در هر فایل میرسیم و با استفاده از کدی مشابه زیر میتوان آن ها را رسم کرد.

# ploting Silhouette score vs number of clusters(k)  
# the maximum shows the optimal k for the kmeans model  
fig, ax = plt.subplots(1,1, figsize =(8,6))  
ax.plot(range(2,25),cost[2:25],"bo-")  
ax.set\_xlabel('k')  
ax.set\_ylabel('Silhouette score')  
ax.set\_title("Silhouette score curve")  
plt.savefig("Silhouette score curve.png")

البته میتوان مقدار هزینه را از روش Elbow با جایگزینی خط کد زیر با مشابهش در کدهای بالا بدست آورد.

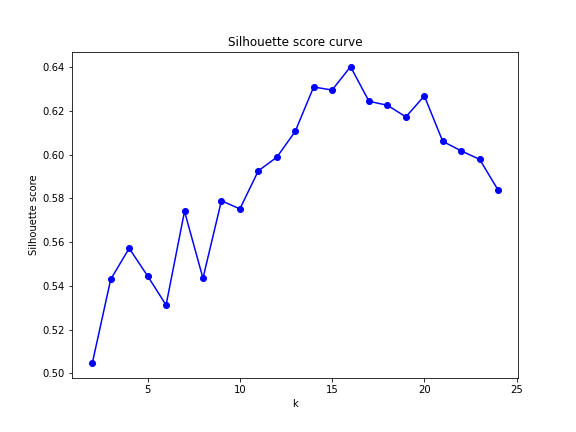
cost[k] = model.summary.trainingCost

پس از رسم نمودار مربوط به این شیوه ارزیابی مشاهده می شود که این روش نمودارهای نزولی ای مشابه تصویر زیر بدست می دهد که نقاطی که منحنی درآنها زاویه دار شده و یا نقطه ای که تقریبا اندازه خطا ثابت شده را میتوان به عنوان k بهینه در نظرگرفت که برای مثال در این نمودار حوالی 16 و 17 است.

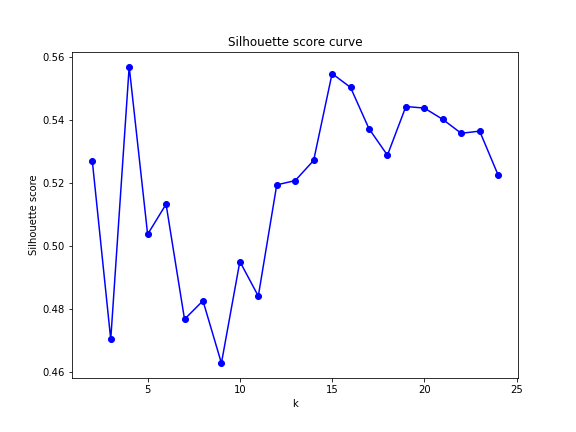


اما از آنجا که محاسبه k بهینه در روش Silhouette ساده تر است و فقط نیاز به ماکزیمم گیری دارد از این روش برای کل حالت ها استفاده شده و در نهایت با رسم نمودار ها به ترتیب برای فایل های متنی به نتایج زیر دست میابیم.

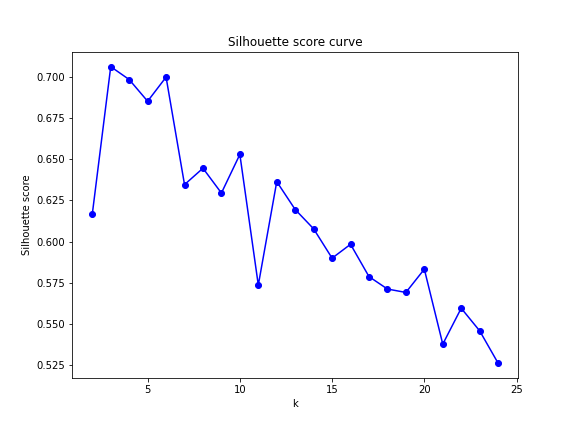
C1: نمودار روش kmeans++ برای فایل اول:



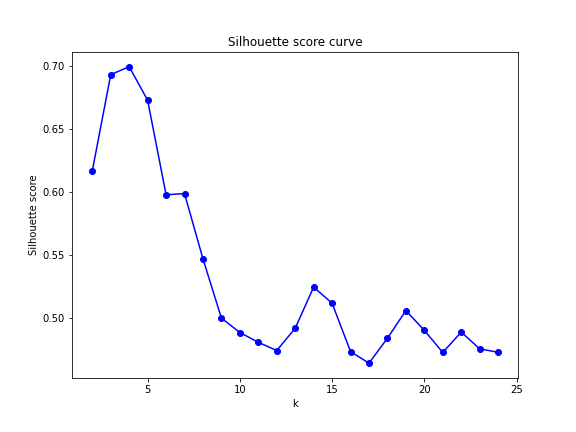
C1: روش Bisecting K-means برای فایل اول:



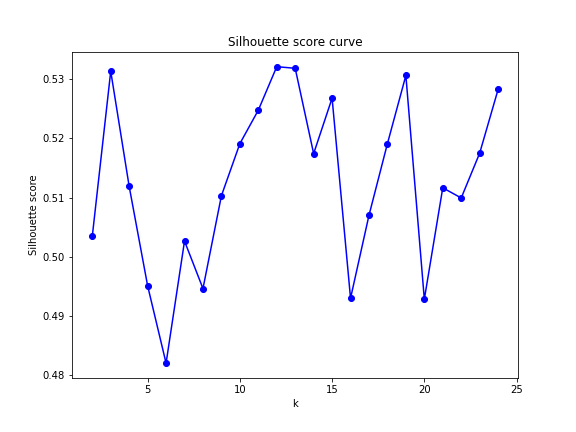
C2: نمودار روش kmeans++ برای فایل دوم:



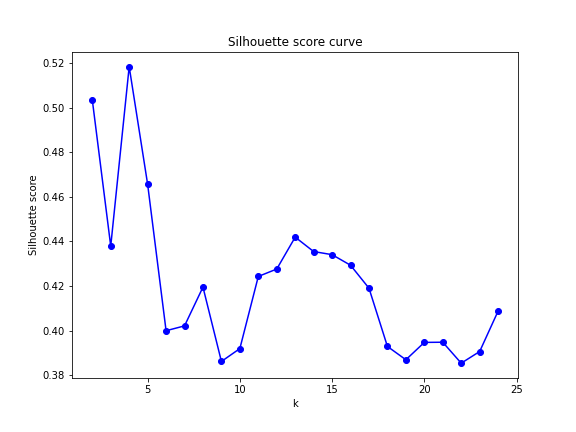
C2: روش Bisecting K-means برای فایل دوم:



C3: نمودار روش kmeans++ برای فایل سوم:



C3: روش Bisecting K-means برای فایل سوم:



ب)

در این بخش میبایست تعداد بهینه خوشه ها را بدست آوریم این کار با استفاده از نمودار های قسمت قبل به راحتی قابل انجام است زیرا در روش Silhouette بهترین تعداد خوشه همان است که مقدار امتیاز Silhouette آن بیشینه شده است و در واقع همان نقطه ماکزیمم نمودار میباشد پس برای این منظور از کد زیر برای همه حالتهای مختلف استفاده میکنیم

k = cost.argmax()  
print("best k for clustering: ", k)

مقدار بهینه k برای هر بخش به این صورت است

برای فایل اول روش kmeans++ مقدار k=16

برای فایل اول روش Bisecting K-means مقدار k=4

برای فایل دوم روش kmeans++ مقدار k=3

برای فایل دوم روش Bisecting K-means مقدار k=4

برای فایل سوم روش kmeans++ مقدار k=12

برای فایل سوم روش Bisecting K-means مقدار k=4

ج)

برای هر فایل روش خوشه بندی ای بهتر است که تعداد بهینه خوشه آن مقدار امتیاز Silhouette بیشتری دارد، با توجه به این نکته و نمودارهای بخش های قبل برای تمام فایل ها در این بخش از روش kmeans++ استفاده میکنیم. البته محاسبات مربوط به این بخش برای همه فایل ها و روش Bisecting K-means در نوت بوک موجود میباشد.

برای بدست آوردن مراکز کلاسترهای برای هر فایل از کد زیر استفاده میکنیم.

kmeans = KMeans(initMode='k-means||').setK(k).setSeed(1).setFeaturesCol("features")  
model = kmeans.fit(df\_kmeans)  
centers = model.clusterCenters()  
  
print("Cluster Centers: ")  
for center in centers:  
 print(center)

که در این کد بار دیگر مدل را با تعداد خوشه k اجرا کرده و مراکز را در لیست centers ذخیره میکنیم و نمایش میدهیم که در نهایت برای هریک از فایل ها نتیجه زیر را داریم.

برای فایل اول با 16 کلاستر داریم:

Cluster Centers:

[738928.3238342 430720.18393782]

[508472.61237785 219457.78501629]

[289311.39145907 703451.08540925]

[458746.55401662 642691.57617729]

[245168.32397959 493590.78571429]

[383938.31740614 363885.19453925]

[389803.93859649 511164.44736842]

[498527.15755627 778321. ]

[199636.85714286 314493.9047619 ]

[541997.53313253 386822.64759036]

[661783.55952381 263828.71130952]

[650725.57931034 756255.30344828]

[768162.48504983 576619.66445183]

[623400.07713499 621188.47658402]

[297558.31476323 276901.95821727]

[546289.03289474 517146.85197368]

برای فایل دوم و 3 کلاستر داریم:

Cluster Centers:

[15.42522253 7.20756677]

[32.36707314 16.54471542]

[11.01487806 22.77268288]

و برای فایل سوم و 12 کلاستر داریم:

Cluster Centers:

[24.01666677 27.32222225]

[13.20200012 17.59400002]

[27.4673913 7.7086957]

[14.94318177 25.54772732]

[15.67624998 11.82875001]

[23.07249999 14.42874994]

[ 5.31666666 14.404762 ]

[29.23620684 18.80862065]

[20.17777782 20.94629627]

[ 8.21304347 22.03913042]

[10.28333322 7.6541667 ]

[19.03478253 5.22173915]

د)

در این بخش پایانی میخواهیم مدت زمان اجرای عملیات خوشه بندی با هر دو روش و با استفاده از تعداد بهینه خوشه و برای هر 3 فایل را با در نظر گرفتن تعداد 1 تا 4 هسته پردازشگر بدست آورده و مقایسه کنیم، برای این کار از کد زیر برای حالات مختلف استفاده میکنیم.

#changing the number of processors for exprimenting the process time of kmeans++ training  
  
T=[]  
  
print("best k for clustering: ", k)  
  
num\_cores= [ 1,2,3,4]  
for i in num\_cores:  
   
 conf = SparkConf()  
 conf.setMaster(f"local[{i}]").setAppName("word-counts") # set the number of cores  
 sc = SparkContext.getOrCreate(conf=conf) #The spark context object is assigned to a variable sc.  
 print(conf.getAll())  
  
 t = time.time()  
 kmeans = KMeans(initMode='k-means||').setK(k).setFeaturesCol("features")  
 model = kmeans.fit(df\_kmeans)  
 print(time.time() - t)  
 T.append(time.time() - t)

در این کد مشابه کدی که در سوال اول زده بودیم عمل میکنیم و به همان طریق تعداد هسته های پردازشگر را تغییر میدهیم. و پس از اعمال روش خوشه بندی زمان اجرای عملیات را در لیست T ذخیره میکنیم. همچنین تعداد هسته ها نیز در لیست num\_cores ذخیره شده است. خروجی کد بالا برای یک نمونه به این صورت است

best k for clustering: 16

[('spark.app.submitTime', '1672647824991'), ('spark.app.name', 'word-counts'), ('spark.submit.pyFiles', ''), ('spark.submit.deployMode', 'client'), ('spark.ui.showConsoleProgress', 'true'), ('spark.master', 'local[1]')]  
1.5731873512268066

[('spark.app.submitTime', '1672647824991'), ('spark.app.name', 'word-counts'), ('spark.submit.pyFiles', ''), ('spark.submit.deployMode', 'client'), ('spark.ui.showConsoleProgress', 'true'), ('spark.master', 'local[2]')]  
1.3014400005340576

[('spark.app.submitTime', '1672647824991'), ('spark.master', 'local[3]'), ('spark.app.name', 'word-counts'), ('spark.submit.pyFiles', ''), ('spark.submit.deployMode', 'client'), ('spark.ui.showConsoleProgress', 'true')]  
1.1812496185302734

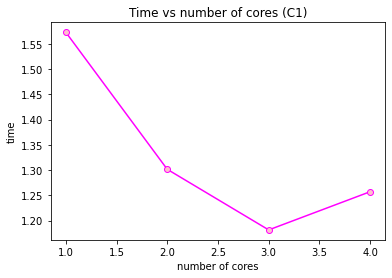
[('spark.app.submitTime', '1672647824991'), ('spark.master', 'local[4]'), ('spark.app.name', 'word-counts'), ('spark.submit.pyFiles', ''), ('spark.submit.deployMode', 'client'), ('spark.ui.showConsoleProgress', 'true')]  
1.2570278644561768

پس از بدست آوردن لیست زمان ها برای هر حالت میتوان با استفاده از کد زیر نمودار مربوط به آن را رسم کرد.

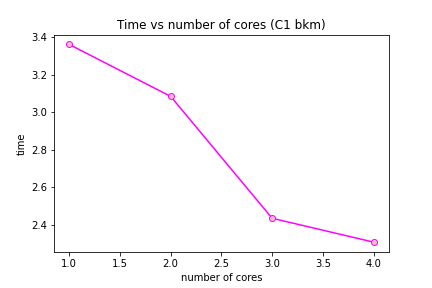
import matplotlib.pyplot as plt  
  
plt.plot(num\_cores,T, color='magenta', marker='o',mfc='pink' ) #plot the data  
  
plt.ylabel('time') #set the label for y axis  
plt.xlabel('number of cores') #set the label for x-axis  
plt.title("Time vs number of cores (C1)") #set the title of the graph  
plt.show() #display the graph  
plt.savefig("Time vs number of cores\_C1.png")

پس از رسم نمودارها نتایج به صورت زیر بدست می آیند.

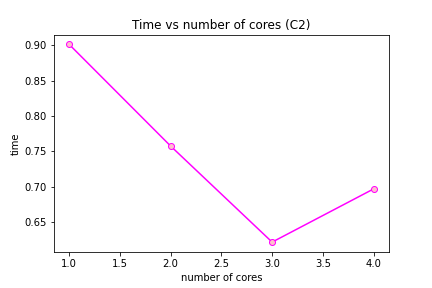
C1: نمودار روش kmeans++ برای فایل اول با k= 16:



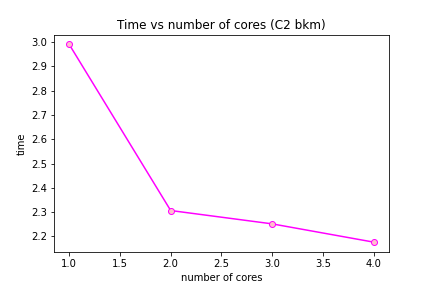
C1: روش Bisecting K-means برای فایل اول با k= 4:



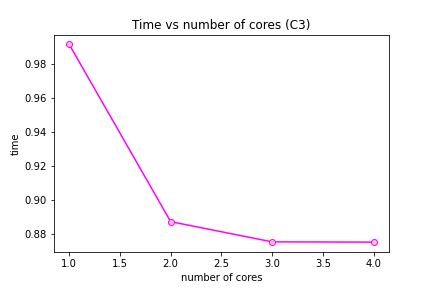
C2: نمودار روش kmeans++ برای فایل دوم با k= 3:



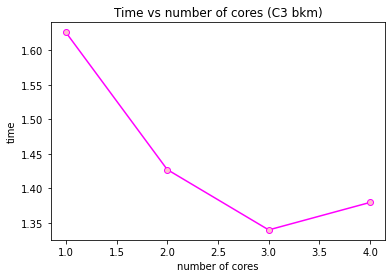
C2: روش Bisecting K-means برای فایل دوم با k= 4:



C3: نمودار روش kmeans++ برای فایل سوم با k= 12:



C3: روش Bisecting K-means برای فایل سوم با k= 4:



باتوجه به نمودارهای به دست آمده میتوان متوجه شد که هرچه تعداد هسته های پردازشگر را افزایش دهیم سرعت محاسبات بیشتر شده و زمان کاهش می یابد و همچنین با توجه به مقادیر نمودارها به نظر میرسد که روش Bisecting K-means در مجموع زمان بیشتری را برای محاسبات نسبت به kmeans++ میخواهد و درواقع روش کندتری است.