# demo5_workbook-KMeans-optimizations

September 24, 2022

## 1 Demo 5 MASTER SOLUTION- K-Means Clustering in Spark

**MIDS w261: Machine Learning at Scale | UC Berkeley School of Information | FALL 2022**

By the end of this demo you should be able to:
* … **implement** K-Means in Spark. * … **explain** how the centroid initialization affects K-Means results & time to convergence. * … **explain** how different join operations are implemented in Spark * … **explain** the challenges of implementing the A Priori algorithm at Scale

### 1.0.1 Notebook Set-Up

[1]:
```
import os
DATA_BUCKET = os.getenv('DATA_BUCKET','')[:-1] # our private storage bucket␣
 ↪location
DEMO05_FOLDER = f"{DATA_BUCKET}/notebooks/jupyter/LiveSessionMaterials/
 ↪wk05Demo_Kmeans"
print(f"Personal Data bucket:  {DATA_BUCKET}")
!gsutil ls -lh  {DEMO05_FOLDER}
```

[2]:
```
# imports
import sys
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import pdb
from functools import reduce as rd
from sklearn.decomposition import PCA
%reload_ext autoreload
%autoreload 2
```

## 2 Content Review: Kmeans

K-Means clustering is an algorithm designed to group examples into K distinct groups based on their features. K-Means clustering can be used to categorize data when we do not have any concrete information about what those categories may look like. For example we may want to perform a market segmentation analysis on our customers but we don't actually know what the segments are.

At a high level clustering is an attempt to group objects in such a way as to make those objects similar to other objects within the group and dissimilar to those outside of the group. In order to perform K-Means clustering we want to minimize the distance between all of the examples that are a part of a cluster and the center of that cluster, also called the *cluster centroid*. The clustering algorithm solves the following cost minimization:

$$\arg\min_{c} \sum_{i=1}^{k} \sum_{x \in c_i} \|x - \mu_i\|^2$$

It turns out that the above minimization problem is very difficult to solve (t falls into the class of problems known as NP-Hard). The K-Means algorithm attempts to solve this clustering problem iteratively by repeating the 2-step process of marking each example as belonging to a single cluster by finding the closest cluster centroid and then adjusting each of the K cluster centroids so that each centroid is in the middle of all the examples that belong to that centroid.

Here is a simple visualization of the process. There are a number of up front choices that must be made before the algorithm can be implemented:

- **K**: the total number of centroids is selected using some apriori information about the desired outcome
- **Distance Function**: typically the euclidian distance is used but this can be any function $d(x, \mu)$
- **Convergence Criteria**: a rule used to determine when the iterative process can stop.

The formal algorithm can then be defined as follows. First, decide the number of clusters $K$. Then:

| Step | Description | Pseudocode |
|---|---|---|
| 1 | Initialize the center ('centroid') of the clusters | ${ }_i = $ random value $, i = 1, ..., k$ |
| 2 | Attribute the closest cluster to each data point | $c_i = \{j : d(x_j, \mu_i) \leq d(x_j, \mu_l), l \neq i, j = 1, ..., n\}$ |
| 3 | Update centroid to the mean of points in that cluster | $\mu_i = \frac{1}{\|c_i\|} \sum_{j \in c_i} x_j, \forall i$ |
| 4 | Repeat steps 2-3 until convergence criteria met | |
| | Notation: | ${ |

## 2.1 Load Demo Data for K-means

The Iris dataset was used in R.A. Fisher's classic 1936 paper, The Use of Multiple Measurements in Taxonomic Problems. It includes three iris species with 50 samples each as well as some properties about each flower. One flower species is linearly separable from the other two, but the other two are not linearly separable from each other.

The columns in this dataset are: > `Id,SepalLengthCm,SepalWidthCm,PetalLengthCm,PetalWidthCm,Species`

If you want to learn more about this dataset visit the UCI Iris Data Set page. Use the cells below to download, preprocess and visualize this data set.

[4]:
```
# download the iris dataset
!curl -L -O https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
 ↪data
# a few data cleaning steps... (remove blank last line, & separate features␣
 ↪from labels)
!head -n -1 iris.data | gsutil cp - {DEMO05_FOLDER}/data/iris_full.csv
!gsutil cat {DEMO05_FOLDER}/data/iris_full.csv | cut -d ',' -f 1-4 |gsutil cp -␣
 ↪{DEMO05_FOLDER}/data/iris_features.csv
!rm iris.data
```

```
mkdir: cannot create directory `data': File exists
```

[5]:
```
# take a look at the first few lines
!gsutil cat {DEMO05_FOLDER}/data/iris_full.csv | head
```

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

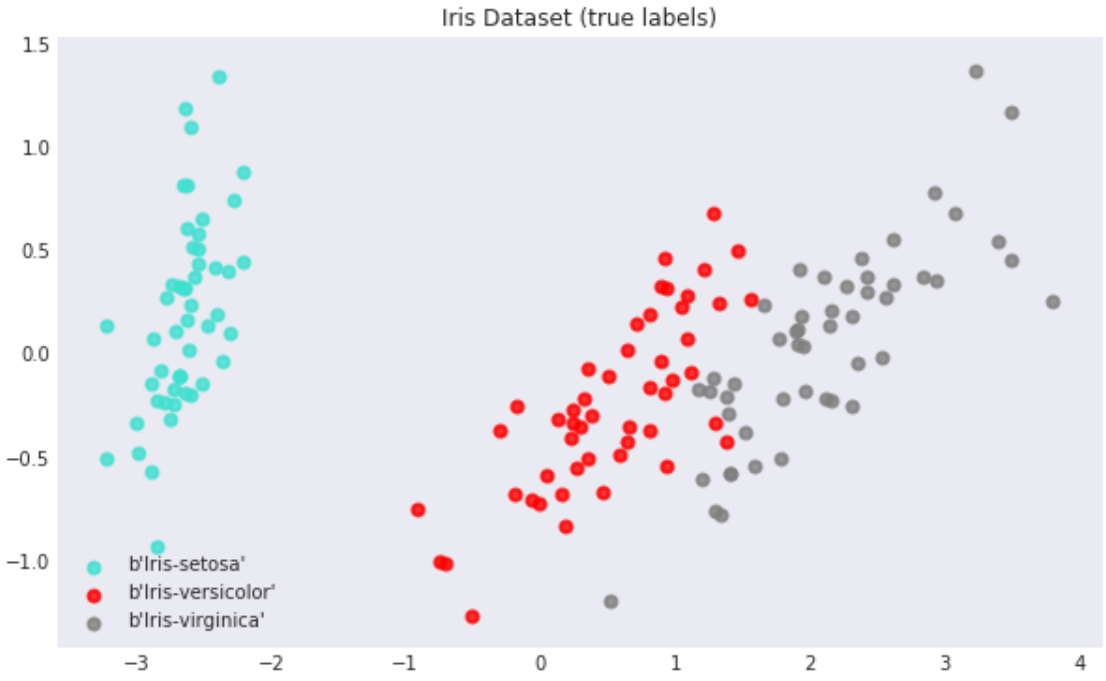
[5]:
```
# set up constants for ease of directory use
IRIS_DATA_WITH_LABELS = f'{DEMO05_FOLDER}/data/iris_full.csv'
IRIS_DATA = f'{DEMO05_FOLDER}/data/iris_features.csv'
```

### 2.2 Visualize Demo Data for Kmeans

For purposes of visualization of this dataset we use a PCA decomposition to project the 4 dimensional iris dataset down to 2 dimensions. This will give us the ability to roughly visualize the effectiveness of our K-Means clustering.

[10]:
```
# read in labeled data
mapping = {b'Iris-setosa': 0,
           b'Iris-versicolor': 1,
           b'Iris-virginica': 2}
names = mapping.keys()
data = np.loadtxt(IRIS_DATA_WITH_LABELS, delimiter=',',
                  converters = {4: lambda s: mapping[s]})
```

[11]:
```
# custom plotting function
from utils import plot_iris_data
plot_iris_data(data[:,0:-1], data[:,-1], names, 'Iris Dataset (true labels)')
```




# 3 K-Means in Python

## 3.1 Implementation

A simple (non-scalable) implementation in Python. For this example we will use the Iris dataset without labels and a K of 3 since we know ahead of time there are 3 different species labels (setosa, versicolor, and virginica).

[487]:
```
# read in the data from the IRIS_DATA file
samples = np.loadtxt(IRIS_DATA,
                     delimiter=',')

# define the number of clusters
k = 3

# define the distance function as the normalized distance
def distance(a, b, ax=1):
    return np.linalg.norm(a - b, axis=ax)

def closestCluster(sample, centers):
    distances = distance(sample, centers)
    return np.argmin(distances)

# define the convergence criteria
convergenceCriteria = .001
```

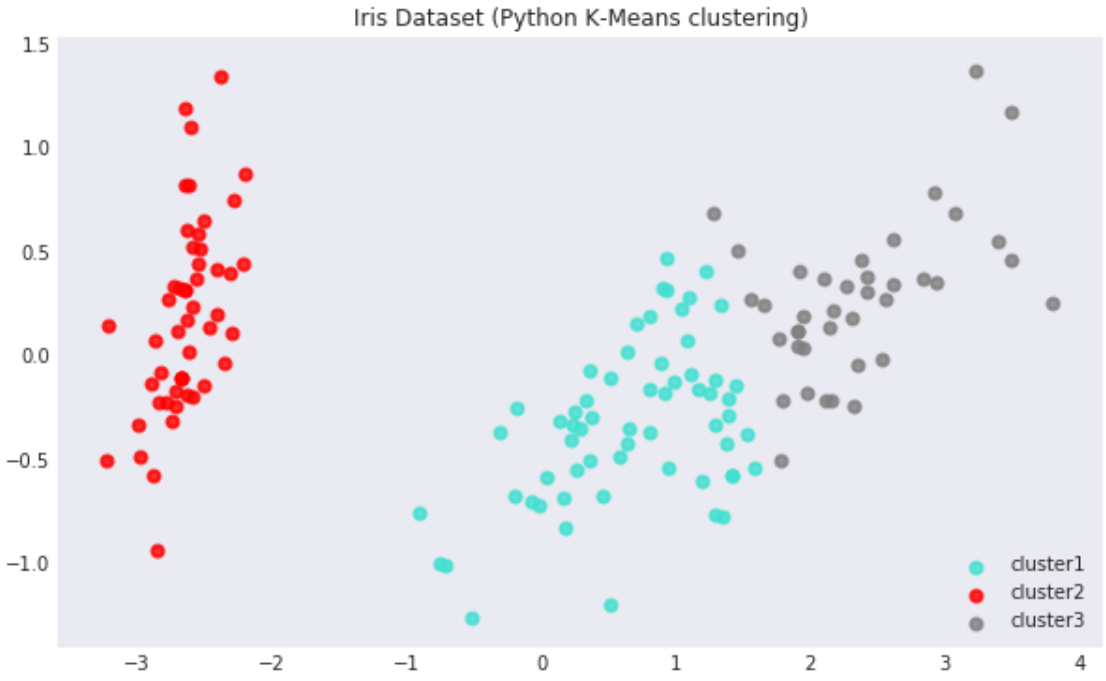
```
# (step1) set the k initial clusters by assigning a random point
centroids = samples[np.random.choice(len(samples), size=k, replace=False)]

while True:
    # (step2) attribute the closest cluster to each point
    clusters = [closestCluster(sample, centroids)
                for sample
                in samples]

    # (step3) set the position of each cluster to the mean of all data points␣
 ↪belonging to that cluster
    last_centroids = np.copy(centroids)
    for i in range(k):
        points = [samples[j]
                  for j
                  in range(len(samples)) if clusters[j] == i]
        centroids[i] = np.mean(points, axis=0)

    # (step4) repeat steps 2-3 until convergence criteria met
    error = distance(centroids,
                     last_centroids,
                     None)
    if error < convergenceCriteria:
        break

print('-------Final Centroids-------')
print(centroids)
```

```
-------Final Centroids-------
[[5.88360656 2.74098361 4.38852459 1.43442623]
 [5.006      3.418      1.464      0.244     ]
 [6.85384615 3.07692308 5.71538462 2.05384615]]
```

### 3.2 Results

Examine the results by assigning each of the datapoints in the file to the nearest cluster and then plot all of the samples along with their labels.

[488]:
```
labels = list(map(lambda sample: closestCluster(sample, centroids), samples))
plot_iris_data(np.array(samples), np.array(labels), ['cluster1', 'cluster2',␣
 ↪'cluster3'],
               'Iris Dataset (Python K-Means clustering)')
```




## 4 K-Means in Spark

This is boilerplate code required for a notebook in this enviroment to create a local spark context.

[6]:
```
from pyspark.sql import SparkSession

app_name = "kmeans_demo_1"
master = "local[*]"
spark = SparkSession\
        .builder\
        .appName(app_name)\
        .master(master)\
        .getOrCreate()

sc = spark.sparkContext
```

[7]:
```
sc.getConf().getAll()
```

[7]:
```
[('spark.app.name', 'kmeans_demo'),
 ('spark.rdd.compress', 'True'),
 ('spark.serializer.objectStreamReset', '100'),
 ('spark.driver.port', '34339'),
 ('spark.master', 'local[*]'),
 ('spark.executor.id', 'driver'),
```

```
 ('spark.submit.deployMode', 'client'),
 ('spark.app.id', 'local-1569541057213'),
 ('spark.ui.showConsoleProgress', 'true'),
 ('spark.driver.host', 'docker.w261')]
```

### 4.1 Implementation

Compare this Spark K-Means implementation to the original, non-scalable implementation in Python. Structurally they are quite similar.

[14]:
```
!ls -alh {IRIS_DATA}
```

```
-rw-r--r-- 1 root root 2.4K Sep 10 11:53
/media/notebooks/LiveSessionMaterials/wk05Demo_Kmeans/data/iris_features.csv
```

[ ]:
```
lines = sc.textFile(IRIS_DATA) # 150 data points
samples = lines.map(lambda line: np.array([float(x) for x in line.split(',')]))
samples.first()
```

[9]:
```
%%time
import numpy as np

lines = sc.textFile(IRIS_DATA)
samples = lines.map(lambda line: np.array([float(x) for x in line.split(',')])).
 ↪cache()

# define the number of clusters
k = 3

# define the distance function as the normalized distance
def distance(a, b):
    return np.linalg.norm(a - b)

def closestCluster(p, centers):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(centers)):
        tempDist = distance(p, centers[i])
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex

# define the convergence criteria
convergenceCriteria = .001

# (step1) set the k initial clusters by assigning a random point
```

```
centroids = samples.takeSample(False, k, 1)

while True:
    # (step2) attribute the closest cluster to each point
    clusters = samples.map(lambda p: (closestCluster(p, centroids), (p, 1)))

    newpoints = clusters.reduceByKey(lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0],␣
 ↪p1_c1[1] + p2_c2[1])) \
                        .map(lambda st: (st[0], st[1][0] / st[1][1])) \
                        .collect()

    error = sum(np.sum(distance(centroids[iK], p)) for (iK, p) in newpoints)

    # (step3) set the position of each cluster to the mean of all data points␣
 ↪belonging to that cluster
    for (iK, p) in newpoints:
        centroids[iK] = p

    # (step4) repeat steps 2-3 until convergence criteria met
    if error < convergenceCriteria:
        break
print("-newpoints-")
print(newpoints)
print('-------Final Centroids-------')
print(np.array(centroids))
```

```
-newpoints-
[(2, array([5.006, 3.418, 1.464, 0.244])), (0, array([5.9016129 , 2.7483871 ,
4.39354839, 1.43387097])), (1, array([6.85      , 3.07368421, 5.74210526,
2.07105263]))]
-------Final Centroids-------
[[5.9016129  2.7483871  4.39354839 1.43387097]
 [6.85       3.07368421 5.74210526 2.07105263]
 [5.006      3.418      1.464      0.244     ]]
CPU times: user 170 ms, sys: 40 ms, total: 210 ms
Wall time: 2.07 s
```

### 4.2 GroupByKey

[15]:
```
%%time

lines = sc.textFile(IRIS_DATA)
samples = lines.map(lambda line: np.array([float(x) for x in line.split(',')])).
 ↪cache()

def addFunc(a,b):
    return a+b
```

```
# define the number of clusters
k = 3

# define the distance function as the normalized distance
def distance(a, b):
    return np.linalg.norm(a - b)

def closestCluster(p, centers):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(centers)):
        tempDist = distance(p, centers[i])
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex

# define the convergence criteria
convergenceCriteria = .001

# (step1) set the k initial clusters by assigning a random point
centroids = samples.takeSample(False, k, 1)

clusters = samples.map(lambda p: (closestCluster(p, centroids), p))

clusters.groupByKey()\
                .map(lambda x: (x[0], rd(addFunc, x[1])/len(x[1])))\
                .collect()
```

```
CPU times: user 40 ms, sys: 20 ms, total: 60 ms
Wall time: 746 ms
```

[12]:
```
%%time
newpoints
```

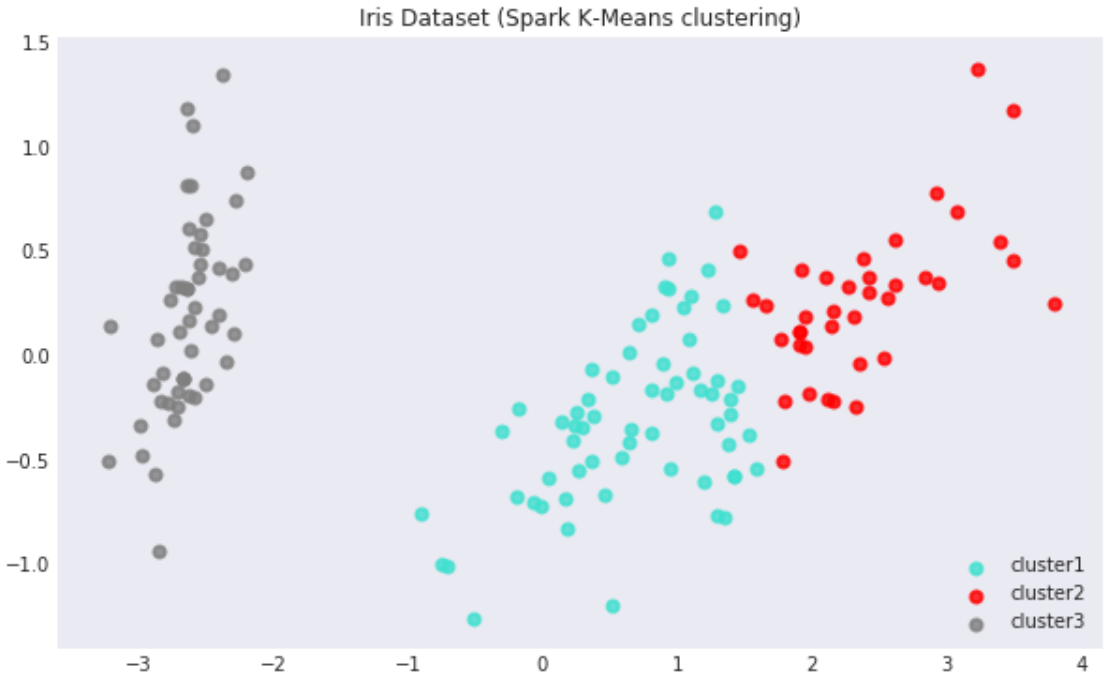
```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 44.3 µs
```

[12]:
```
[(2, array([5.03684211, 3.29824561, 1.70350877, 0.34385965])),
 (0, array([6.33846154, 2.89450549, 5.0010989 , 1.70659341])),
 (1, array([6.3 , 3.35, 5.8 , 2.45]))]
```

### 4.3 Results

Examine the results by assigning each of the datapoints in the file to the nearest cluster and then plot all of the samples along with their labels.

[14]:
```
samples = np.loadtxt(IRIS_DATA,
                     delimiter=',')
labels = list(map(lambda sample: closestCluster(sample, centroids),
              samples))
plot_iris_data(np.array(samples),
     np.array(labels),
     ['cluster1', 'cluster2', 'cluster3'],
     'Iris Dataset (Spark K-Means clustering)')
```




Discussion

How does changing the centroid initialization affect the result? How does changing the convergence criteria affect the result?

### 4.3.1 See also:

- KMeans++
- KMeans||
- Canopy clustering

## 5 Accumulators

Definitive Guide book, pg. 241

Accumulators are Spark's equivalent of Hadoop counters. Like broadcast variables they represent shared information across the nodes in your cluster, but unlike broadcast variables accumulators

are *write-only* … in other words you can only access their values in the driver program and not on your executors (where transformations are applied). As convenient as this sounds, there are a few common pitfalls to avoid. Let's take a look.

Run the following cell to create a sample data file representing a list of `studentID, courseID, final_grade`…

[7]:
```
grades_csv_str = """10001,101,98
10001,102,87
10002,101,75
10002,102,55
10002,103,80
10003,102,45
10003,103,75
10004,101,90
10005,101,85
10005,103,60"""

grades_csv_loc = f'{DEMO05_FOLDER}/grades.csv'


!echo "{grades_csv_str}" | gsutil cp - {grades_csv_loc}
```

```
Overwriting data/grades.csv
```

Suppose we want to compute the average grade by course and student while also tracking the number of failing grades awarded. We might try something like this:

[204]:
```
# function to increment the accumulator as we read in the data
def parse_grades(line, accumulator):
    """Helper function to parse input & track failing grades."""
    student,course,grade = line.split(',')
    grade = int(grade)
    if grade < 65:
        accumulator.add(1)
    return(student,course, grade)
```

[205]:
```
# <--- SOLUTION --->
# function to increment the accumulator as we read in the data
def parse_grades(line, accumulator):
    """Helper function to parse input & track failing grades."""
    student,course,grade = line.split(',')
    grade = int(grade)
    return(student,course, grade)
```

[205]:
```
# initialize an accumulator to track failing grades
nFailing = sc.accumulator(0)
```

[206]:
```
# compute averages in spark

gradesRDD = sc.textFile('data/grades.csv')\
              .map(lambda x: parse_grades(x, nFailing))

studentAvgs = gradesRDD.map(lambda x: (x[0], (x[2], 1)))\
                       .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))\
                       .mapValues(lambda x: x[0]/x[1])

courseAvgs = gradesRDD.map(lambda x: (x[1], (x[2], 1)))\
                      .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))\
                      .mapValues(lambda x: x[0]/x[1])
```

[206]:
```
# <--- SOLUTION --->
# compute averages in spark
# initialize an accumulator to track failing grades
nFailing = sc.accumulator(0)

def accAgg(row):
    grade = row[2]
    if grade < 65:
        nFailing.add(1)

gradesRDD = sc.textFile('data/grades.csv')\
              .map(lambda x: parse_grades(x, nFailing))

gradesRDD.cache()
gradesRDD.foreach(accAgg)


studentAvgs = gradesRDD.map(lambda x: (x[0], (x[2], 1)))\
                       .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))\
                       .mapValues(lambda x: x[0]/x[1])

courseAvgs = gradesRDD.map(lambda x: (x[1], (x[2], 1)))\
                      .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))\
                      .mapValues(lambda x: x[0]/x[1])
```

[207]:
```
# take a look
print("===== average by student =====")
print(studentAvgs.collect())
print("===== average by course =====")
print(courseAvgs.collect())
print("===== number of failing grades awarded =====")
print(nFailing)
```

```
===== average by student =====
```

```
[('10001', 92.5), ('10004', 90.0), ('10002', 70.0), ('10003', 60.0), ('10005',
72.5)]
===== average by course =====
[('102', 62.333333333333336), ('101', 87.0), ('103', 71.66666666666667)]
===== number of failing grades awarded =====
3
```

> **DISCUSSION QUESTIONS:** * What is wrong with the results? (**HINT:** *how many failing grades are there really?*) * Why might this be happening? (**HINT:** *How many actions are there in this code? Which parts of the DAG are recomputed for each of these actions?*) * What one line could we add to the code to fix this problem? * What could go wrong with our "fix"? * How could we have designed our parser differently to avoid this problem in the first place?

### 5.1 Custom Accumulators

https://spark.apache.org/docs/latest/rdd-programming-guide.html#accumulators

While SparkContext supports accumulators for primitive data types like int and float, users can also define accumulators for custom types by providing a custom AccumulatorParam object.

We may want to utilize custom accumulators later in the course when we implement PageRank, or Shortest Path (graph) algorithms

[12]:
```
from pyspark.accumulators import AccumulatorParam

# Spark only implements Accumulator parameter for numeric types.
# This class extends Accumulator support to the string type.
class StringAccumulatorParam(AccumulatorParam):
    def zero(self, value):
        return value
    def addInPlace(self, val1, val2):
        return val1 +" -> "+ val2
```

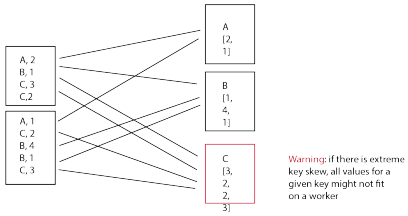
## 6 Aggregations

[125]:
```
from IPython.display import Image
Image(filename="aggregations.png")
```

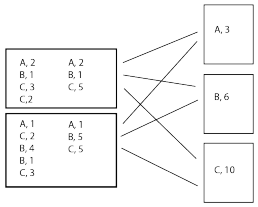
[125]:

A few common aggreation methods for Spark RDDs.

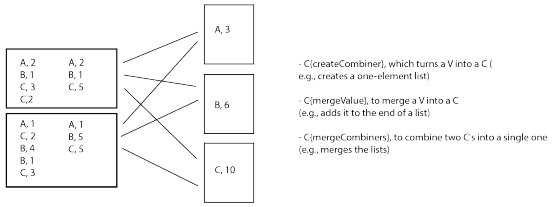
groupByKey()
https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.groupByKey




reduceByKey(Func)
https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.reduceByKey



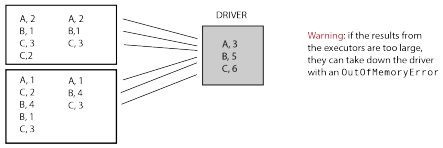

combineByKey(createCombiner, mergeValue, mergeCombiners)
https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.combineByKey




aggregateByKey(zeroValue, seqOp, combOp)
https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.aggregateByKey

start value,
seqOp -> within partition function,
combOp -> across partition function

ex: aggregateByKey(0,max,add)




foldByKey(zeroValue, Func)
https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.foldByKey

Calls combineByKey, but allows us to use a zero value which can be added to the result an arbitrary number of times, and must not change the result (eg. 0 for addition, 1 for multiplication)

treeAggregate(zeroValue, seqOp, combOp, depth)
https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.treeAggregate

Same as aggregate except it "pushes down" some of the subaggregations (creating a tree from executor to executor) before performing final aggregations on the driver.

What if we wanted to get a list of letter grades that each student recieved as well as their average?

[1]:
```
def toLetterGrade(x):
    if x > 92.0:
        return "A"
    elif x > 82.0:
        return "B"
    elif x > 72.0:
        return "C"
    elif x > 65.0:
        return "D"
    else:
        return "F"
```

```
def getCounts(a,b):
    return (a[0] + b[0], a[1] + b[1], toLetterGrade(a[0])+toLetterGrade(b[0]))
```

[ ]:
```
# THIS WILL THROW AN ERROR ON PURPOSE
studentAvgs = gradesRDD.map(lambda x: (x[0], (x[2], 1)))\
                       .reduceByKey(getCounts)\
                       .mapValues(lambda x: ((x[0]/x[1]),x[2]))\
                       .collect()
```

[15]:
```
from IPython.display import Image
Image(filename="Bob-Ross-3.jpg")
```

[15]:


[212]:
```
# gradesRDD.map(lambda x: (x[0], (x[2], 1))).collect()
gradesRDD.map(lambda x: (x[0], (x[2], 1))).reduceByKey(getCounts).collect()
```

[212]:
```
[('10001', (185, 2, 'AB')),
 ('10004', (90, 1)),
 ('10002', (210, 3, 'AC')),
 ('10003', (120, 2, 'FC')),
 ('10005', (145, 2, 'BF'))]
```

### 6.0.1 foldByKey allows us to specify a zero value

[213]:
```
gradesRDD.map(lambda x: (x[0], (x[2], 1))).foldByKey((0,0,""),getCounts).
↪collect()
```

[213]:
```
[('10001', (185, 2, 'AB')),
 ('10004', (90, 1, 'FB')),
 ('10002', (210, 3, 'AC')),
 ('10003', (120, 2, 'FC')),
 ('10005', (145, 2, 'BF'))]
```

#### 6.0.2 <— SOLUTION —>

**SOLUTION**
Student 10004 is assigned an initial 'F' grade. We have not met the foldByKey requirement because our zeroValue changes the result

#### 6.0.3 Can we solve this problem using a combineByKey which provides more granular control over the parameters

https://backtobazics.com/big-data/apache-spark-combinebykey-example/

[214]:
```
def createCombiner(a):
    return a

def mergeValues(a,b):
    return (a[0] + b[0], a[1] + b[1], toLetterGrade(a[0])+toLetterGrade(b[0]));

def mergeCombiners(a,b):
    return (a[0] + b[0], a[1] + b[1], toLetterGrade(a[0])+toLetterGrade(b[0]))

studentAvgs = gradesRDD.map(lambda x: (x[0], (x[2], 1)))\
                       .combineByKey(createCombiner,mergeValues,mergeCombiners)\
                       .mapValues(lambda x: ((x[0]/x[1]),x[2]))
```

[215]:
```
gradesRDD.map(lambda x: (x[0], (x[2], 1))).
 ↪combineByKey(createCombiner,mergeValues,mergeCombiners).collect()
```

[215]:
```
[('10001', (185, 2, 'AB')),
 ('10004', (90, 1)),
 ('10002', (210, 3, 'AC')),
 ('10003', (120, 2, 'FC')),
 ('10005', (145, 2, 'BF'))]
```

#### 6.0.4 <— SOLUTION —>

**SOLUTION**
We know that `reduceByKey` calls `combineByKey` under the hood. We have not changed anything here, so the result is exactly the sameas in our first attempt when we used `reduceByKey`

#### 6.0.5 aggragateByKey requires a null and start value as well as two different functions. One to aggregate within partitions, and one to aggregate across partitions

[216]:
```
def seqOp(a,b):
    return(a[0] + b[0], a[1] + b[1], a[2]+toLetterGrade(b[2]))

def combOp(a,b):
    return (a+b);
```

[217]:
```
letterAccum = sc.accumulator("===", StringAccumulatorParam())

gradesRDD.foreach(lambda x: letterAccum.add(toLetterGrade(x[2])))

gradesRDD.map(lambda x: (x[0], (x[2], 1, x[2])))\
         .aggregateByKey((0,0,""),seqOp,combOp)\
         .mapValues(lambda x: ((x[0]/x[1]),x[2]))\
         .collect()
```

[217]:
```
[('10001', (92.5, 'AB')),
 ('10004', (90.0, 'B')),
 ('10002', (70.0, 'CFC')),
 ('10003', (60.0, 'FC')),
 ('10005', (72.5, 'BF'))]
```

[193]:
```
print (letterAccum)
```

```
=== -> === -> F -> C -> B -> B -> F -> === -> A -> B -> C -> F -> C
```

## 7 Joins

- join
  https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.join
- leftOuterJoin
  https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.leftOuterJoin
- rightOuterJoin
  https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.rightOuterJoin
- fullOuterJoin
  https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.fullOuterJoin
- cartesian
  https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.cartesian

[229]:
```
x = sc.parallelize([("a", 1), ("b", 4)])
y = sc.parallelize([("a", 2), ("c", 8)])
```

```
sorted(x.fullOuterJoin(y).collect())
#[('a', (1, 2)), ('b', (4, None)), ('c', (None, 8))]
```

[229]:
```
[('a', (1, 2)), ('b', (4, None)), ('c', (None, 8))]
```

[230]:
```
sorted(x.rightOuterJoin(y).collect())
```

[230]:
```
[('a', (1, 2)), ('c', (None, 8))]
```

[231]:
```
sorted(x.leftOuterJoin(y).collect())
```

[231]:
```
[('a', (1, 2)), ('b', (4, None))]
```

Lets load some data for the code examples

[148]:
```
person = spark.createDataFrame([
    (0, "Bill Chambers", 0, [100]),
    (1, "Matei Zaharia", 1, [500, 250, 100]),
    (2, "Michael Armbrust", 1, [250, 100])])\
  .toDF("id", "name", "graduate_program", "spark_status")
graduateProgram = spark.createDataFrame([
    (0, "Masters", "School of Information", "UC Berkeley"),
    (2, "Masters", "EECS", "UC Berkeley"),
    (1, "Ph.D.", "EECS", "UC Berkeley")])\
  .toDF("id", "degree", "department", "school")
sparkStatus = spark.createDataFrame([
    (500, "Vice President"),
    (250, "PMC Member"),
    (100, "Contributor")])\
  .toDF("id", "status")
```

[433]:
```
# run as is
joinExpression = person["graduate_program"] == graduateProgram['id']
```

[434]:
```
# run as is
wrongJoinExpression = person["name"] == graduateProgram["school"]
```

[482]:
```
# run as is
person.join(graduateProgram, joinExpression).show()
```

```
+---+----------------+----------------+---------------+---+-------+------------
-------+-----------+
| id|            name|graduate_program|   spark_status| id| degree|
department|     school|
+---+----------------+----------------+---------------+---+-------+------------
-------+-----------+
|  0|   Bill Chambers|               0|          [100]|  0|Masters|School of
Informa…|UC Berkeley|
```

```
|  1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|
EECS|UC Berkeley|
|  2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|
EECS|UC Berkeley|
+---+----------------+----------------+---------------+---+-------+------------
-------+-----------+
```

[181]:
```
person.join(graduateProgram, wrongJoinExpression).show()
```

```
+---+----+---------------+------------+---+------+----------+------+
| id|name|graduate_program|spark_status| id|degree|department|school|
+---+----+---------------+------------+---+------+----------+------+
+---+----+---------------+------------+---+------+----------+------+
```

[522]:
```
# Spark perfoms an "inner" join by default. But we can specify this explicitly.
# Try different join types.
joinType = "outer"
joinType = "left_outer"
joinType = "right_outer"
```

[523]:
```
person.join(graduateProgram, joinExpression, joinType).show()
```

```
+----+----------------+---------------+---------------+---+-------+------------
--------+-----------+
|  id|            name|graduate_program|   spark_status| id| degree|
department|     school|
+----+----------------+---------------+---------------+---+-------+------------
--------+-----------+
|   0|   Bill Chambers|               0|          [100]|  0|Masters|School of
Informa…|UC Berkeley|
|   1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|
EECS|UC Berkeley|
|   2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|
EECS|UC Berkeley|
|null|            null|            null|           null|  2|Masters|
EECS|UC Berkeley|
+----+----------------+---------------+---------------+---+-------+------------
--------+-----------+
```

### 7.0.1 Which keys do outer joins evaluate?

### 7.0.2 A departure from traditional joins:

[185]:
```
gradProgram2 = graduateProgram.union(spark.createDataFrame([
    (0, "Masters", "Duplicated Row", "Duplicated School")]))
gradProgram2.createOrReplaceTempView("gradProgram2")
```

[143]:
```
# Think of left semi joins as filters on a DataFrame, as opposed to the␣
 ↪function of a conventional join
joinType = "left_semi"
gradProgram2.join(person, joinExpression, joinType).show()
```

```
+---+-------+--------------------+----------------+
| id| degree|          department|          school|
+---+-------+--------------------+----------------+
|  0|Masters|School of Informa…|     UC Berkeley|
|  0|Masters|      Duplicated Row|Duplicated School|
|  1|  Ph.D.|                EECS|     UC Berkeley|
+---+-------+--------------------+----------------+
```

[187]:
```
gradProgram2.show()
```

```
+---+-------+--------------------+----------------+
| id| degree|          department|          school|
+---+-------+--------------------+----------------+
|  0|Masters|School of Informa…|     UC Berkeley|
|  2|Masters|                EECS|     UC Berkeley|
|  1|  Ph.D.|                EECS|     UC Berkeley|
|  0|Masters|      Duplicated Row|Duplicated School|
+---+-------+--------------------+----------------+
```

[142]:
```
joinType = "left_anti"
gradProgram2.join(person, joinExpression, joinType).show()
```

```
+---+-------+----------+-----------+
| id| degree|department|     school|
+---+-------+----------+-----------+
|  2|Masters|      EECS|UC Berkeley|
+---+-------+----------+-----------+
```

### 7.0.3 Natural Joins

**DANGER**: Natural joins make implicit guesses at the columns on which you would like to join. Why is this bad?

### 7.0.4 Cross (Cartesian) Joins

Or, Cartesian products. Cross joinsare inner joins that do not specify a predicate. Cross joins will join every single row in the left DataFrame with every single row in the right DataFrame

[562]:
```
joinType = "cross"
graduateProgram.join(person, joinExpression, joinType).show()
```

```
+---+-------+--------------------+-----------+---+---------------+------------
---+---------------+
| id| degree|          department|     school| id|
name|graduate_program|   spark_status|
+---+-------+--------------------+-----------+---+---------------+------------
---+---------------+
|  0|Masters|School of Informa…|UC Berkeley|  0|   Bill Chambers|
0|          [100]|
|  1|  Ph.D.|                EECS|UC Berkeley|  1|   Matei Zaharia|
1|[500, 250, 100]|
|  1|  Ph.D.|                EECS|UC Berkeley|  2|Michael Armbrust|
1|     [250, 100]|
+---+-------+--------------------+-----------+---+---------------+------------
---+---------------+
```

[146]:
```
person.crossJoin(graduateProgram).show()
```

```
+---+----------------+---------------+---------------+---+-------+------------
-------+-----------+
| id|            name|graduate_program|   spark_status| id| degree|
department|     school|
+---+----------------+---------------+---------------+---+-------+------------
-------+-----------+
|  0|   Bill Chambers|              0|          [100]|  0|Masters|School of
Informa…|UC Berkeley|
|  0|   Bill Chambers|              0|          [100]|  2|Masters|
EECS|UC Berkeley|
|  0|   Bill Chambers|              0|          [100]|  1|  Ph.D.|
EECS|UC Berkeley|
|  1|   Matei Zaharia|              1|[500, 250, 100]|  0|Masters|School of
Informa…|UC Berkeley|
|  1|   Matei Zaharia|              1|[500, 250, 100]|  2|Masters|
EECS|UC Berkeley|
|  1|   Matei Zaharia|              1|[500, 250, 100]|  1|  Ph.D.|
EECS|UC Berkeley|
|  2|Michael Armbrust|              1|     [250, 100]|  0|Masters|School of
Informa…|UC Berkeley|
|  2|Michael Armbrust|              1|     [250, 100]|  2|Masters|
EECS|UC Berkeley|
```

```
|  2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|
EECS|UC Berkeley|
+---+----------------+----------------+---------------+---+-------+------------
-------+-----------+
```

**DANGER**: How many rows would we end up with from a cross join if each table had 1000 rows?

# 8 A Priori

(see WK5 slides)

# 9 ————————- END NOTEBOOK ————————————

## 9.1 Future work: Clustering for anomaly detection

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.323.6870&rep=rep1&type=pdf
https://proquest-safaribooksonline-com.libproxy.berkeley.edu/book/databases/9781491972946
https://towardsdatascience.com/best-clustering-algorithms-for-anomaly-detection-d5b7412537c8

## 9.2 KDD Cup 1999 Data Set

For each connection, the data set contains information like the number of bytes sent, login attempts, TCP errors, and so on. Each connection is one line of CSV-formatted data set, containing 38 features, like this:

```
0,tcp,http,SF,215,45076,
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,
0.00,0.00,0.00,0.00,1.00,0.00,0.00,0,0,0.00,
0.00,0.00,0.00,0.00,0.00,0.00,0.00,normal.
```

This connection, for example, was a TCP connection to an HTTP service—215 bytes were sent and 45,706 bytes were received. The user was logged in, and so on. Many features are counts, like num_file_creations in the 17th column.

Many features take on the value 0 or 1, indicating the presence or absence of a behavior, like su_attempted in the 15th column. They look like the one-hot encoded categorical features from Chapter 4, but are not grouped and related in the same way. Each is like a yes/no feature, and is therefore arguably a categorical feature. It is not always valid to translate categorical features as numbers and treat them as if they had an ordering. However, in the special case of a binary categorical feature, in most machine learning algorithms, mapping these to a numeric feature taking on values 0 and 1 will work well.

The rest are ratios like dst_host_srv_rerror_rate in the next-to-last column, and take on values from 0.0 to 1.0, inclusive.

Interestingly, a label is given in the last field. Most connections are labeled normal., but some have been identified as examples of various types of network attacks. These would be useful in learning to distinguish a known attack from a normal connection, but the problem here is anomaly detection and finding potentially new and unknown attacks. This label will be mostly set aside for our purposes.

[ ]: