To compile the code:
At the prompt in the project's home folder type:
$ sbt assembly

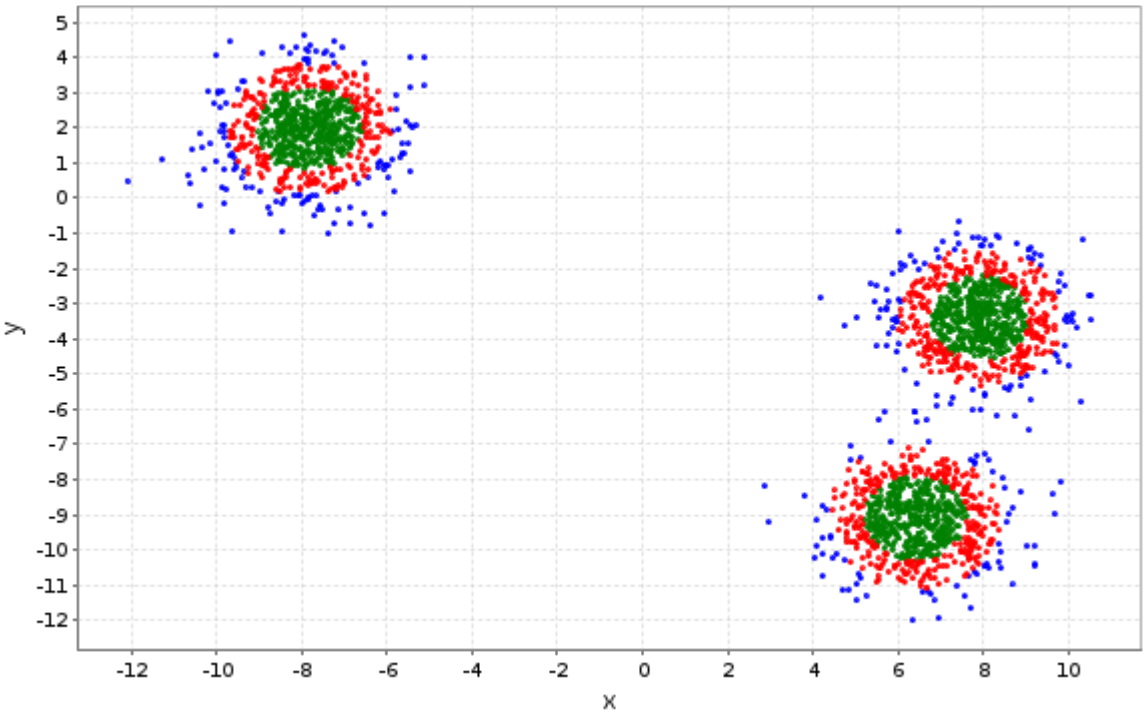
To run the program:
At the prompt in the project's home folder type:

spark-submit --class Spectral --master local[*] target/scala-2.10/PIClustering-assembly-1.0.jar <k-value> <max-iterations> <gamma-value> <dataset-file path>

Note: the program needs all 4 parameters

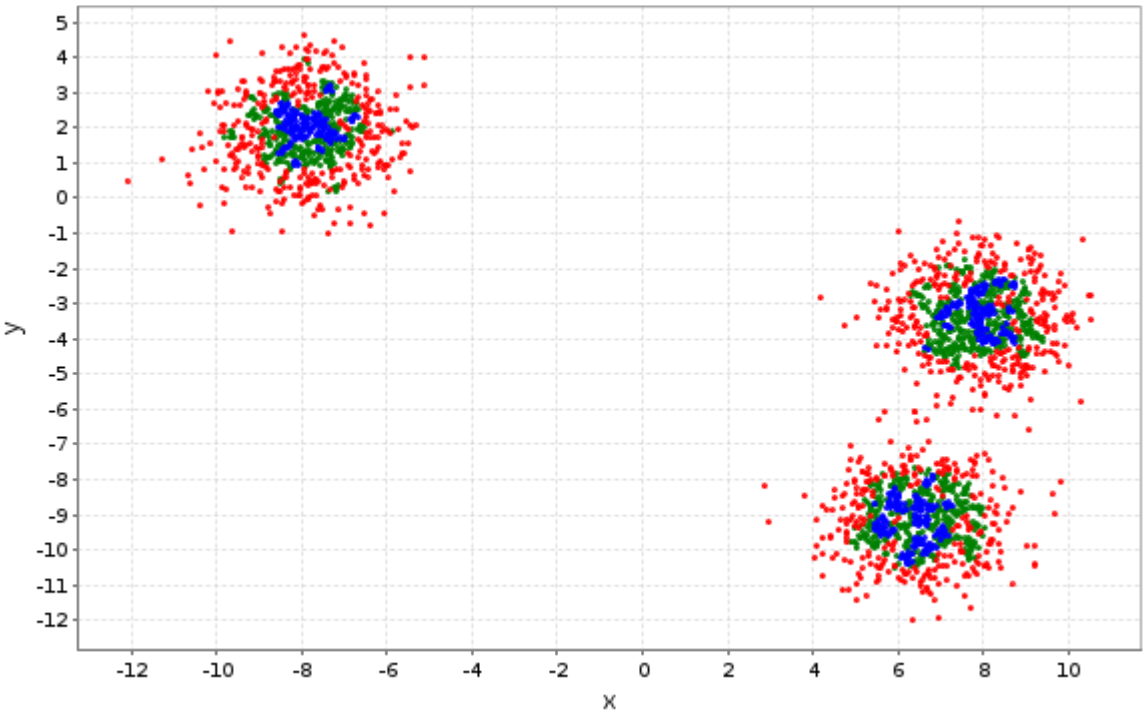
1. k=3, x=10, g=1, blobs.txt
Runtime = 10 secs
Scatter plot of clusters:

2.
k=3,
x=10,



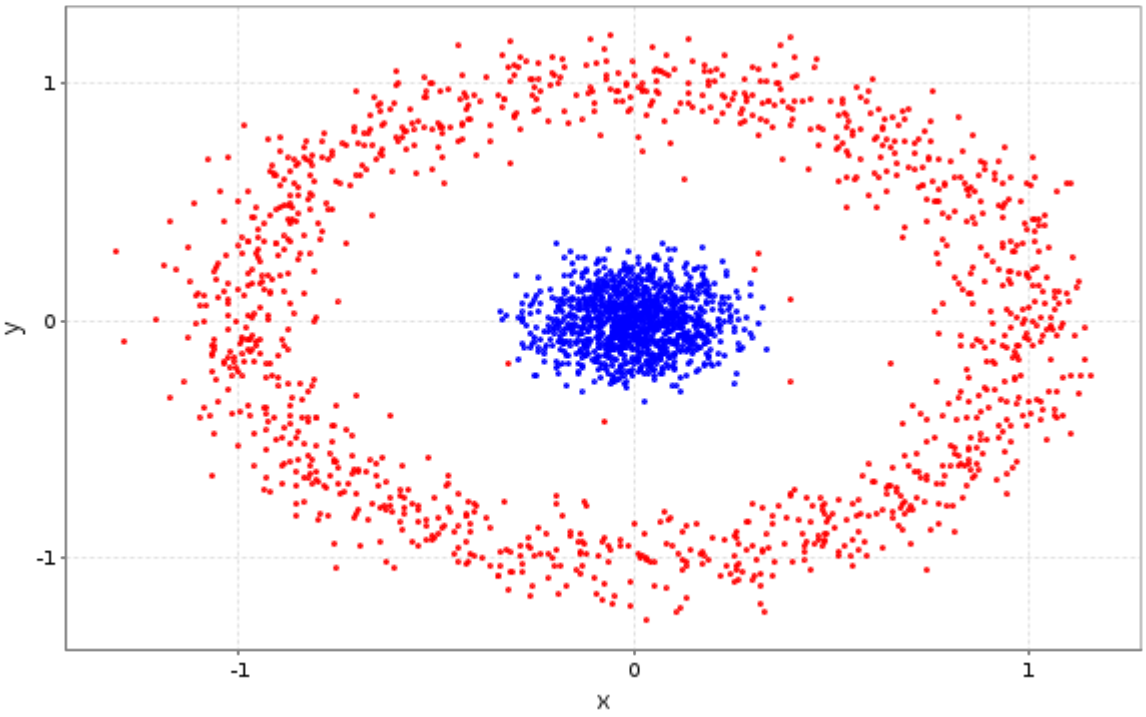
g=100, blobs.txt
Runtime = 9 secs
Scatter plot of clusters:

3.
k=2,
x=10,
g=50,



circles.txt
Runtime = 9 secs
Scatter plot of clusters:

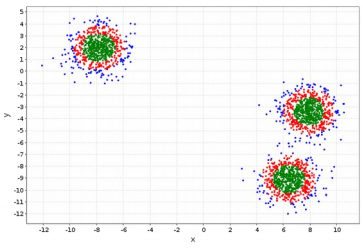
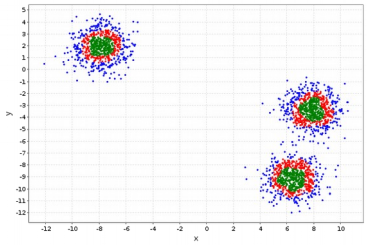
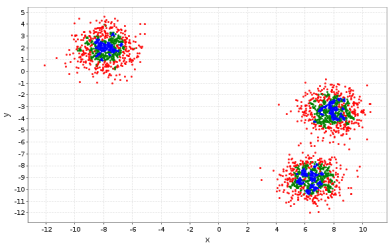
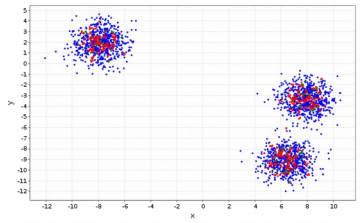
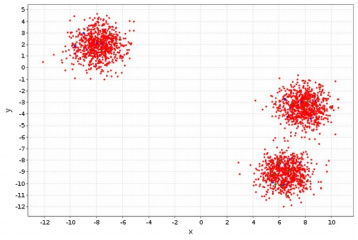
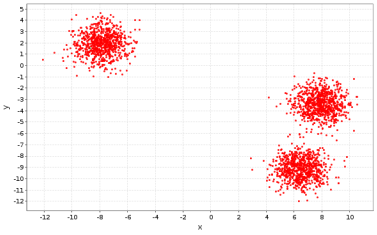
4.
k=2,
x=10,
g=50,



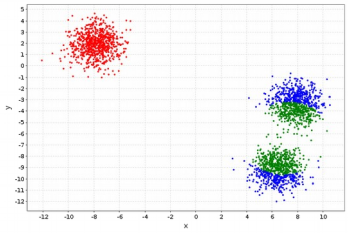
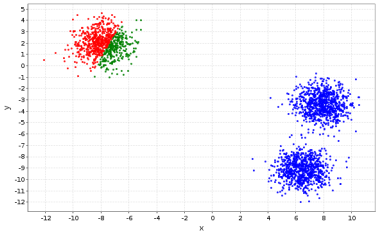
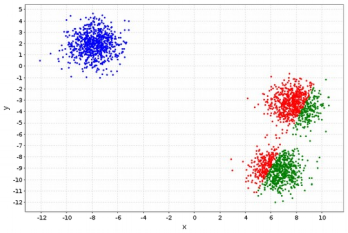
moons.txt
Crashes during Power iteration clustering. Out of memory even with max iterations = 1
Tried on azure but fails with some null pointer exception sometimes or stops responding.

5. As the value of g increases, the clustering begins to diminish, that is, all the points start to become

part of the same cluster.
Following are the scatter plots for g values 1, 10, 100, 1000, 10000, 100000 respectively

On the other hand, decreasing g further from 1, gives different results for different g values. Following are results for g values 0.1, 0.01, 0.001. It's also difficult to see what actually is happening in the PIC for these g values as the outputs seem unpredictable.

6.
The run times for the first three runs seem to be constant(unaffected by any of the parameters/data file). However it seems from the partial runs on the moons.txt file that there is a bottleneck in the PowerIterationClustering.run() method in some sum operation where usually the program crashes. It could be because of the vector-matrix multiplication($v_{t+1} = M \cdot v_t$) where the vector has dimensions same as the number of data points.

Bonus 2.1
The following chunk of Spark code assumes that the affinity matrix has been computed without filtering with the i != j or i < j predicates, for instance, in the following way:

```
val affinityMatrix = indexedDataPoints.cartesian(indexedDataPoints)
            .map({
                case (((indXi), (xi)), ((indXj), (xj))) => (indXi, indXj,
                    Math.exp(-g.value * (Math.pow(xj(0).toDouble - xi(0).toDouble,
2) + Math.pow(xj(1).toDouble - xi(1).toDouble, 2))))
            })
```

Computation of $L = D^{-1/2}AD^{-1/2}$:

```
val diagonal = affinityMatrix.groupBy({ case (r, c, value) => r })
            .map({ case (r, v) => (r, v.foldLeft(0.0)({ case (sumRow, (r, c, v)) =>
sumRow + v })) })

/*pre-multiply affinity matrix by the diagonal matrix of row sums.
Here each element of the affinity matrix simply needs to be multiplied by
the value of the diagonal matrix having the same row number as the element*/

//prepare affinity matrix with row as key and column, value as value for joining
with diagonal matrix
        val rowKeyedAffinityMatrix = affinityMatrix.map({case(r, c, v) => (r, (c,
v))})
        //join and map product to the row, column of the elements
        val da = rowKeyedAffinityMatrix.join(diagonal).map({case(r, ((c, v),
rowSum)) => (r, c, Math.pow(rowSum, -0.5) *v)})

/*post-multiply the resulting matrix by the diagonal matrix of row sums.
Here each element of the matrix 'da' simply needs to be multiplied by
the value of the diagonal matrix having the same column number as the element*/

//prepare affinity matrix with column as key and row, value as value for joining
with diagonal matrix
        val colKeyedDA = da.map({case(r, c, v) => (c, (r, v))})
        //join and map product to the row, column of the elements
        val laplacian = colKeyedDA.join(diagonal).map({case(c, ((r, v), rowSum)) =>
(r, c, Math.pow(rowSum, -0.5) *v)})
```