MapReduce on Windows Azure

Daytona – K-means algorithm user guide

Table of Contents

1 ALGORITHM

K-means clustering is an algorithm to classify or to group data (typically consisting of well-formed records/objects) based on attributes/features into K (positive integer) number of groups/clusters. Suppose we have the following sample CSV dataset and need to classify it into two (K = 2) clusters.

Medicine Name	Manufacturer	Weight Index	pH Value
Medicine A	Company1	1	1
Medicine B	Company2	2	1
Medicine C	Company1	4	3
Medicine D	Company3	5	4

The algorithm will perform the following steps.

1. Pick the columns (should be numeric) on which to compute coordinates of records. In the sample dataset, let's use 'Weight Index' and 'pH Value'. The coordinates of all the records in the sample dataset will thus be a 2-D vector.
2. Randomly pick two records as the initial set of clusters. In the sample dataset, let's use 'Medicine A' and 'Medicine B'.
   Cluster-1 = (1, 1)
   Cluster-2 = (2, 1)
3. Calculate the distance between each record and the clusters. In the sample dataset, let's use [Euclidean distance](#) as we are using 2D vector co-ordinates. The distance for first iterations will be

$$\mathbf{D}^0 = \begin{bmatrix} 0 & 1 & 3.61 & 5 \\ 1 & 0 & 2.83 & 4.24 \end{bmatrix} \quad \begin{matrix} \mathbf{c}_1 = (1,1) & group-1 \\ \mathbf{c}_2 = (2,1) & group-2 \end{matrix} \quad \begin{bmatrix} 1 & 2 & 4 & 5 \\ 1 & 1 & 3 & 4 \end{bmatrix} \begin{matrix} \text{Weight Index} \\ \text{pH Value} \end{matrix}$$

$$A \quad B \quad C \quad D$$

   Each column in the distance matrix symbolizes a record. The first row of the distance matrix corresponds to the distance of each object to the first centroid and the second row is the distance of each object to the second centroid. For example, distance from medicine C = (4, 3) to the first centroid $\mathbf{c}_1 = (1,1)$ is $\sqrt{(4-1)^2+(3-1)^2} = 3.61$, and its distance to the second centroid $\mathbf{c}_2 = (2,1)$ is $\sqrt{(4-2)^2+(3-1)^2} = 2.83$, etc.
4. Associate each record with one of the clusters based on the minimum distance. In the sample dataset, associations look as below:
   Medicine A -> Cluster-1
   Medicine B -> Cluster-2
   Medicine C -> Cluster-2
   Medicine D -> Cluster-2
5. Compute the new co-ordinates of the clusters by taking the average of the co-ordinates of the records associated with them. In the sample dataset, the new cluster coordinates are:

$$\mathbf{c}_1 = (1,1)$$

$$\mathbf{c}_2 = (\frac{2+4+5}{3}, \frac{1+3+4}{3}) = (\frac{11}{3}, \frac{8}{3})$$

6. If the new co-ordinates of the clusters have changed, then repeat steps 3 to 6. Otherwise (convergence criteria met) report the cluster co-ordinates as the final result.

1.1 SAMPLE IMPLEMENTATION USING DAYTONA

1.1.1 ABSTRACT

Considering a CSV dataset (already available in Azure Blob Storage) having two numerical columns (similar to the ones shown in previous section), K-means execution in Daytona can be broken into the following steps.

1. Write a 'Controller' which is responsible for composing a 'job' to compute the clusters and submitting it iteratively till the convergence criteria is met. It should also determine the initial clusters by randomly picking records from the input dataset and determining their coordinates based on the 'cluster columns' selected by the user. Alternatively, it can directly take the coordinates of initial clusters as an input from the user.
2. Write a 'DataPartitioner' for reading the input dataset from Azure Blob Storage and then uniformly splitting/partitioning it.
3. Write a 'Mapper' for determining the coordinates of each input record in a split/partition (based on the 'cluster columns' selected by the user), computing its nearest cluster and producing a key-value pair.
    a. Key – 2D vector representing the cluster coordinates.
    b. Value – 2D vector representing the record coordinates.
4. Write a 'Combiner' to reduce the output payload of the 'Mapper' by producing a weighted sum of records having the same nearest cluster. (i.e. merge the values having the same key).
5. Write a 'KeyPartitioner' that uniformly splits/partitions the keys output by the 'Combiner' for parallel processing.
6. Write a 'Reducer' for computing the new coordinates of each input cluster by taking the average of the corresponding record coordinates and producing a key-value pair.
    a. Key - 2D vector representing the old coordinates of the cluster.
    b. Value - 2D vector representing the new coordinates of the cluster.
7. Write a 'RecordWriter' to write the final clusters to Azure Blob Storage.

1.1.2 DETAILS

1.1.2.1 Daytona Assemblies

The sample makes use of the following assemblies.

Assembly	*Purpose*
Research.MapReduce.Core.dll	Contains the base classes, interfaces for implementing the different components (mapper, reducer, writer etc.) required to run a job and API for creating and submitting jobs.
Research.MapReduce.Library.dll	Contains commonly used implementation of data partitioning, key partitioning and output writing.

1.1.2.2 Controller

The Controller is responsible for validating the input parameters, formulating the job definition and iteratively submitting it for execution till the convergence criteria is met.

1.1.2.2.1 PARAMETER VALIDATION

Parameter	***Parameter Type***	***Description***	***Requirement***	***Validation***
StorageConnectionString	string	Connection string required to connect to the Azure storage used for input/output.	Mandatory	Valid azure storage connection string format.
InputDataLocation	string	URI pointing to the blob containin the input data in CSV format.	Mandatory	Valid blob URI format. Should exist in the storage account specified by the parameter 'StorageConnectionString'.
OutputDataLocation	string	Name of the blob container to which clusters will be written to.	Mandatory	1. Valid blob container name. 2. If it already exists, then should be accessible so that it can cleaned during initialization.
RequestedNumberOfMappers	int	The number of mappers the caller has requested to use for running a job.	Mandatory	Should be greater than zero.
RequestedNumberOfReducers	int	The number of Reducers the caller has requested to use for running a job.	Mandatory	Should be greater than zero.
ClusterColumns	string	The names of the columns (in CSV format) whose values are to be used as the dimensions of the cluster.	Mandatory	Should be exactly two column names.
InitialClustersLocation	string	URI pointing to the blob containing initial clusters. Should contain the cluster column values in CSV format.	Optional	1. Valid blob URI format. 2. Should exist in the storage account specified by the parameter 'StorageConnectionString'. 3. Should be equivalent to the parameter 'Clusters'.
Clusters	int	Number of clusters to compute. Needs to be specified only if parameter 'InitialClustersLocation' has not been specified.	Optional	Should be greater than zero.
JobTimeoutInMinutes	int	Duration within which a single k-means job must complete.	Mandatory	Should be greater than zero.

1.1.2.2.2 JOB DEFINITION

'BlobCsvPartitioner' (available in 'Research.MapReduce.Library) is used to split the input CSV dataset (available as an Azure Blob) equally among the number of mappers specified by the user.

In the CSV dataset header, the indexes of columns that will be used to compute coordinates of records ('cluster columns') are passed as job parameters (as they are not subject to change during iteration).

```
partitioner = new BlobCsvPartitioner(this.CloudClient, this.InputDataLocation,
this.RequestedNumberOfMappers);


JobConfiguration jobConfig = new JobConfiguration();

jobConfig.MapperType = typeof(KMeansClusteringMapper);

jobConfig.CombinerType = typeof(KMeansClusteringReducer);

jobConfig.MapOutputStorage = MapOutputStoreType.Local;

jobConfig.KeyPartitioner = typeof(HashModuloKeyPartitioner<Point, WeightedPoint>);

jobConfig.ReducerType = typeof(KMeansClusteringReducer);

jobConfig.JobTimeout = TimeSpan.FromMinutes(JobTimeoutInMinutes);

jobConfig.ExceptionPolicy = new TerminateOnFirstException();

jobConfig.JobParameters.Add(ParameterNames.FirstClusterColumnIndex, firstClusterColumnIndex

.ToString());

jobConfig.JobParameters.Add(ParameterNames.SecondClusterColumnIndex, secondClusterColumnIndex

.ToString());

// Use simple iterative task scheduling policy.

jobConfig.TaskSchedulingPolicy = TaskSchedulingPolicy.SimpleIterative;


Job job = new Job(jobConfig, this);

job.DataPartitioner = partitioner;

job.NoOfReduceTasks = this.RequestedNumberOfReducers;

job.RecordWriter = new KMeansClusteringWriter(this.CloudClient, this.OutputDataLocation);

// Use data cache if available.

job.UseCacheIfAvailable = true;

return job;
```

1.1.2.2.3 JOB SUBMISSION, ITERATION AND FINAL RESULT

A serialized string representing the current coordinates of the clusters is passed as a map task parameter (as they are subject to change during iteration). Also, 'job.GetReduceOutputsToController' is set to 'true' so that data output by the reducer can be sent back to the controller to verify the convergence criteria.

The job submission is iteratively done till the clusters computed for the current iteration match with the ones computed during the previous iteration. Once the clusters converge, all of them are written to a single blob as the final result.

```
while (true)
{
     // Add the input clusters for use by mappers.
     kMeansJob.AddOrReplaceMapParameter(ParameterNames.InputClusters, previousClusters.
                    GetString());
     kMeansJob.GetReduceOutputsToController = true;
     kMeansJob.Run();
     numberOfIterations++;
     if (kMeansJob.JobExecutionInfo.State != JobState.Completed)
     {
         // If the job failed, then return.
         return;
     }
     // New clusters will be returned as reduce output.
     newClusters = new List<Point>();
     IEnumerable<IReduceResult<Point, WeightedPoint>> reduceOutput = kMeansJob.
     GetReduceOutputs<Point, WeightedPoint>();
     foreach (KeyValuePair<Point, WeightedPoint> output in reduceOutput.SelectMany(
     tuple => tuple))
     {
        // Value is the new co-ordinates of cluster.
        newClusters.Add(output.Value);
     }

     if (Converged(previousClusters, newClusters))
     {
```

```
        // If the newly computed clusters vary from the ones computed in the
        // previous iteration, then stop the iteration.
        break;
     }
     previousClusters = newClusters;
}

// Write the clusters produced by different reducers into a single blob.
CloudBlobContainer outputContainer = this.CloudClient.BlobClient.GetContainerReference(
   this.OutputDataLocation);
CloudBlob finalClustersBlob = outputContainer.GetBlobReference("finalclusters");
using (StreamWriter sw = new StreamWriter(finalClustersBlob.OpenWrite()))
{
   foreach (Point centroid in newClusters)
   {
      sw.WriteLine("Cluster-{0},{1}", Guid.NewGuid(), centroid);
   }
}
this.Results.Add(ResultNames.FinalClustersBlobUri, finalClustersBlob.Uri.AbsoluteUri);
this.Results.Add(ResultNames.NumberOfIterations, numberOfIterations.ToString());
```

1.1.2.3 Mapper

The mapper retrieves and saves the input parameters (current coordinates of the clusters, index of the 'cluster columns') during configuration.

For each input record ('value'), the mapper creates a numeric representation ('Point') of its cluster columns and determines the nearest input cluster. The nearest input cluster and numerical representation of the record (The 'point' is converted to a 'WeightedPoint' so as to facilitate summation of records having the same nearest cluster) are returned as key and value respectively.

```
public class KMeansClusteringMapper : IMapper<int, string, Point, WeightedPoint>
    {
```

```
        #region Fields


        private List<Point> inputClusters;

        private int firstClusterColumnIndex;

        private int secondClusterColumnIndex;


        #endregion


        public void Configure(MapContext<int, string> context)

        {

            inputClusters = context.TaskParameters[ParameterNames.InputClusters].ToString().GetPoints();

            firstClusterColumnIndex = int.Parse(context.JobParameters[ParameterNames.

                FirstClusterColumnIndex].ToString());

            secondClusterColumnIndex = int.Parse(context.JobParameters[ParameterNames.

                SecondClusterColumnIndex].ToString());

        }


        public IEnumerable<KeyValuePair<Point, WeightedPoint>> Map(int key, string value, MapContext<int,
string> context)

        {

            Point point = value.GetPoint(firstClusterColumnIndex, secondClusterColumnIndex);

            Point nearestCluster = point.GetNearestCentroid(inputClusters);

            yield return new KeyValuePair<Point, WeightedPoint>(nearestCluster, point.

                GetWeightedPoint());

        }

    }
```

For the sample CSV dataset, during first iteration the map outputs will be:

1. Key -> Point (x, y) = (1, 1) and Value -> WeightedPoint (x, y, weight) = (1, 1, 1)
2. Key -> Point (x, y) = (2, 1) and Value -> WeightedPoint (x, y, weight) = (2, 1, 1)
3. Key -> Point (x, y) = (2, 1) and Value -> WeightedPoint (x, y, weight) = (4, 3, 1)
4. Key -> Point (x, y) = (2, 1) and Value -> WeightedPoint (x, y, weight) = (5, 4, 1)

1.1.2.4 REDUCER AND COMBINER

When acting as a combiner, it just makes a sum of all records having the same nearest cluster. When acting as a reducer, after making the sum it also computes the new value for the cluster by calculating the average of all the records closest to it.

```
public class KMeansClusteringReducer : IReducer<Point, WeightedPoint, Point, WeightedPoint>

    {

        public IEnumerable<KeyValuePair<Point, WeightedPoint>> Reduce(Point key,
IEnumerable<WeightedPoint> values, ReduceContext<Point, WeightedPoint> context)

        {

            WeightedPoint valuesSum = new WeightedPoint();

            foreach (WeightedPoint val in values)

            {

                valuesSum.Add(val);

            }

            if (context.CreatedForCombining)

            {

                yield return new KeyValuePair<Point, WeightedPoint>(key, valuesSum);

            }

            else

            {

                yield return new KeyValuePair<Point, WeightedPoint>(key, valuesSum.Average());

            }

        }


        public void Configure(ReduceContext<Point, WeightedPoint> context)

        { }

    }
```

For the sample CSV dataset, during first iteration the combiner outputs will be:

1. Key -> Point (x, y) = (1, 1) and Value -> WeightedPoint (x, y, weight) = (1, 1, 1)
2. Key -> Point (x, y) = (2, 1) and Value -> WeightedPoint (x, y, weight) = (2, 1, 1) + (4, 3, 1) + (5, 4, 1) = (11, 8, 3)

For the sample CSV dataset, during first iteration the reducer outputs will be:

1. Key -> Point (x, y) = (1, 1) and Value -> WeightedPoint (x, y, weight) = Average (1, 1, 1) = (1, 1, 1)
2. Key -> Point (x, y) = (2, 1) and Value -> WeightedPoint (x, y, weight) = Average(11, 8, 3) = (3.6667, 2.6667, 1)

1.1.2.5 WRITER AND RESULT

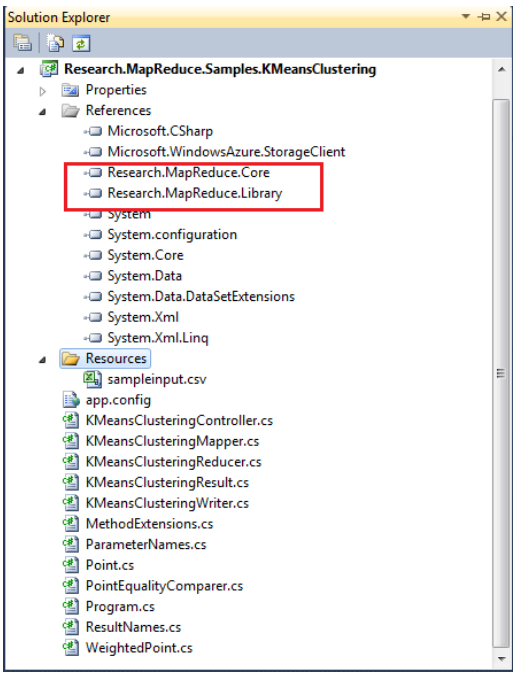
For each key-value pair output by reducer, the writer serializes both the key and the value into a string (CSV containing the 2D vector coordinate values) and writes them on the same line separated by '#'. The writer maintains a cache of 4MB and writes to the Blob Storage only when the cache size is exceeded.

```
public void Write(string outputPartition, IEnumerable<KeyValuePair<Point, WeightedPoint>> records)
        {
            CloudBlobContainer container = this.CloudClient.BlobClient.GetContainerReference(this.
                ContainerName);
            container.CreateIfNotExist();
            blob = container.GetBlobReference(outputPartition);
            buffer = new byte[BufferSize];
            IEnumerator<KeyValuePair<Point, WeightedPoint>> enumerator = records.GetEnumerator();
            WriteToBuffer(enumerator, buffer);
            using (Stream stream = blob.OpenWrite())
            {
                stream.Write(buffer, 0, noOfBytesWrittenInBuffer);
                if (bufferFull)
                {
                    using (StreamWriter sw = new StreamWriter(stream))
                    {
                        // Re-write the current record as the buffer is full.
                        while (enumerator.MoveNext())
                        {
                           sw.Write(record.Key);
                           sw.Write("#");
                           sw.Write(record.Value);
                           sw.WriteLine();
                        }
```

```
                    }
                }
            }
        }

public IReduceResult<Point, WeightedPoint> GetResult()
        {
            if (bufferFull)
            {
                return new KMeansClusteringResult(this.CloudClient, blob.Uri.AbsoluteUri);
            }
            else
            {
                byte[] localBuffer = new byte[noOfBytesWrittenInBuffer];
                Buffer.BlockCopy(buffer, 0, localBuffer, 0, noOfBytesWrittenInBuffer);
                return new KMeansClusteringResult(this.CloudClient, blob.Uri.AbsoluteUri, localBuffer);
            }
        }
```

1.1.3 RUNNING THE SAMPLE



Open the project 'Research.MapReduce.Samples.KMeansClustering.csproj' in a new instance of visual studio 2010. Make sure that the required Daytona assemblies have been referred to in the project from a valid path.

1.1.3.1 DAYTONA RUNTIME DEPLOYMENT

Please refer to the document for 'Daytona-Setup Guide' to deploy the Daytona runtime onto an azure service.

1.1.3.2 UPLOAD INPUT

The input dataset should be in CSV format and must be uploaded to an Azure Blob. The project contains a sample dataset named 'sampleinput.csv' added to the 'Resources'.

1.1.3.3 SUBMIT

All the assemblies corresponding to the project will be uploaded to a dedicated Azure blob container as a single logical unit named 'Package' and then submitted for execution. Following code snippet shows how to create and submit the 'KMeansClustering' sample.

```
if (args.Length == 0)

{

   Console.WriteLine(@"Usage:- -InputDataLocation::<value> -OutputDataLocation::<value> -Mappers::<value>

    -Reducers::<value> -ClusterColumns::<value> -JobTimeoutInMinutes::<value> [-Clusters::<value> | -
```

```
     InitialClustersLocation::<value>]");
   Console.WriteLine("Using default arguments.");
   args = GetSampleInputArgs();
}

Dictionary<string, string> controllerArgs = new Dictionary<string, string>();
foreach (string arg in args)
{
                string[] argArray = arg.Split(new string[] { "::" },
StringSplitOptions.RemoveEmptyEntries);
                controllerArgs.Add(argArray[0].TrimStart('-'), argArray[1]);
}

MapReduceClient client = new MapReduceClient("clusteringpackage", "clustering-" + Guid.NewGuid(),
       typeof(KMeansClusteringController).AssemblyQualifiedName, controllerArgs);
client.Submit(masterConnectionString);
```

The default arguments will use the sample input from ‘Resources’.

```
private static string[] GetSampleInputArgs()
{
    string inputContainerName = "clustering-input-" + Environment.UserName;
    string outputContainerName = "clustering-output-" + Environment.UserName;

    string[] args = new string[6];

    // Arg0 - InputDataLocation
    CloudBlobContainer container = GetInitializedContainer(inputContainerName);
    CloudBlob blob = container.GetBlobReference("sampleinput.csv");
    blob.UploadText(Resources.sampleinput);

    // Arg0 - InputDataLocation
    args[0] = string.Format("-{0}::{1}", ParameterNames.InputDataLocation, blob.Uri);
```

```
    // Arg1 - OutputDataLocation
    container = GetInitializedContainer(outputContainerName);
    args[1] = string.Format("-{0}::{1}", ParameterNames.OutputDataLocation, container.Name);

    // Arg2 - Mappers
    args[2] = string.Format("-{0}::{1}", ParameterNames.Mappers, 4);

    // Arg3 - Reducers
    args[3] = string.Format("-{0}::{1}", ParameterNames.Reducers, 2);

    // Arg4 - Clusters
     args[4] = string.Format("-{0}::{1}", ParameterNames.Clusters, 4);

     // Arg5 - ClusterColumns
     args[5] = string.Format("-{0}::{1}", ParameterNames.ClusterColumns, "Duration,Time");

     return args;
}
```

The following keys in the app.config must be set to appropriate values as explained below.

```
<appSettings>

    <add key="MasterConnectionString"
value="DefaultEndpointsProtocol=http;AccountName=<AccountName>;AccountKey=<AccountKey>"/>

<add key="StorageConnectionString"
value="DefaultEndpointsProtocol=http;AccountName=<AccountName>;AccountKey=<AccountKey>"/>

  </appSettings>
```

- **MasterConnectionString:** This should be the same as the storage account used by the Daytona runtime deployment.
- **StorageConnectionString:** This should be the storage account intended for uploading the input dataset and writing the output of the job(s) as well as the final result. It can be the same as the 'MasterConnectionString' if both accounts belong to the current user. This value is automatically set as the value of the controller parameter 'StorageAccountString' by 'MapReduceClient' which looks for this specific key in the 'appSettings' and sends it as a controller argument irrespective of whether the user has specified any controller arguments or not.