K-MEANS CLUSTERING IN APACHE IGNITE MACHINE LEARNING

Here we show how to use Apache Ignite Machine Learning to do classification using the K-Means Clustering algorithm. The code is in Scala.

If you're new to this, start with our introductory tutorial for Apache Ignite Machine Learning. We've also previously covered K-means clustering, which finds the centers and assigns each set of features to one. See our Python and Spark tutorials.

What is interesting about Ignite queries is they support SQL. That is because Ignite can store all kinds of data. In this example we just put vector data, since that's what machine learning uses. But you can put traditional database data in Ignite as well—that is its true purpose.

The problem with Apache Ignite is there are not too many examples on the internet. The few that are there are too difficult to understand. Apache Ignite does have a user guide, but it's not detailed. The guide points you to examples, but it doesn't explain them. That means you have to look at the JavaDocs to try to figure out how to do things—which is what we've done for you here.

Ignite ML K-means tutorial

The code for this example is here. You don't need to install Apache Ignite in order to run this example. This is because this program will start an Ignite instance by itself.

Download the two_classed_iris.csv data from GitHub here. It's not important to know what that data means. We just need to know that it's a features-label dataset suitable for K-means clustering.

Start Ignite and create the data cache

This code starts an instance of Apache Ignite. It's not necessary to install Ignite. The program downloads the JAR files needed to make it run. Then this Scala program starts it.

(Note: When you start the example below Ignite starts but does not stop. That's because we have started an instance.)

First, we give Ignite two working directories. Note that we use the default constructor **IgniteConfiguration()**.

```
 val PersistencePath = "/Users/walkerrowe/Downloads/ignite"
  val WalPath = "/Users/walkerrowe/Downloads/wal"

 val config = new IgniteConfiguration()

  val ignite = Ignition.start(config)
```

Write data to Ignite

This method creates a cache object into which we will write the data. It's called **cache**, because Ignite calls its database an in-memory cache. (It is a distributed in-memory cache, which is what makes it so powerful.)

```
 val dataCache = getCache(ignite)
```

This code is not necessary to explain. I just copied it from one of their examples and converted it to Scala using IntelliJ. It just created the cache object and makes it ready to receive data.

```
private def getCache(ignite: Ignite) = {
    val cacheConfiguration = new CacheConfiguration
    cacheConfiguration.setName("ML_EXAMPLE_" + UUID.randomUUID)
    cacheConfiguration.setAffinity(new RendezvousAffinityFunction(false, 10))
    ignite.createCache(cacheConfiguration)
  }
```

Read file into Ignite data cache

Now we read the .csv file and form a label-features vector. The data is one label, the first column, and 4 features columns. The data is fixed width with no delimiter other than spaces.

```
0       5.1        3.5        1.4        0.2
0       4.9        3        1.4        0.2
0       4.7        3.2        1.3        0.2
0       4.6        3.1        1.5        0.2
0       5        3.6        1.4        0.2
0       5.4        3.9        1.7        0.4
```

Here we use the Ignite utility **VectorUtils.of(double values ...)** to create a vector. We will use another vector utility to tell Ignite which column is the label and which are the features.

The method **put()** writes the data to the Ignite data cache. The format is **put(key, value)**, where value is, in our example, a Vector.

```
val file =
"/Users/walkerrowe/Documents/igniteSource/ignite/examples/src/main/resources/
datasets/two_classed_iris.csv"


  val bufferedSource = io.Source.fromFile(file)

  var i: Integer = 0
  for (line <- bufferedSource.getLines) {

    val cols: Array = line.split("\\s+").map(_.trim)

    dataCache.put(i, VectorUtils.of(cols(0).toDouble,
      cols(1).toDouble, cols(2).toDouble,
      cols(3).toDouble, cols(4).toDouble))
    i = i + 1
  }
  bufferedSource.close
```

Preprocessing

The call to the training model **trainer.fit(ignite, dataCache, vectorizer)** includes a **vectorizer**.

We use **DummyVectorizer,** which you could call the **do-nothing vectorizing** as it only separates features and labels. **LabelCoordinate.FIRST** is an enumeration (enum) that means take the first column in the Vector as the label.

```
val vectorizer = new
DummyVectorizer().labeled(Vectorizer.LabelCoordinate.FIRST)
```

Ignite has a whole set of preprocessors. Those do the things you would normally do with machine learning data like:

- Extracting features
- Normalizing data (i.e., scaling it)
- Forcing text or real numbers based on a threshold to labels, e.g., 0 or 1
- Replacing missing values with something like the average of other values
- Creating a one-hot vector, which is a special way of encoding categorical data

Create and train the model

Now we train the model. There are a handful of parameters, but we take the default. Then we set the number of clusters to 2 since all the input labels are either 0 or 1.

```
val trainer = new KMeansTrainer()
```

```
  trainer.withAmountOfClusters(2)

  val mdl = trainer.fit(ignite, dataCache, vectorizer)
```

Print the cluster centers

The goal of k-means clusters is to find the centers and to assign each set of features to one of the clusters. So here we print the coordinates that are the centers of the two clusters.

```
 val centers: Array = mdl.getCenters

  for (c <- centers) {
    System.out.print("centers ")
    for (a <- c.asArray()) {
      printf("%.2f ", a)
    }
    System.out.println("\n")
  }
```

Create a cache query and run predictions across that

Now we loop through the data we have written to the Ignite data cache. Then we run a prediction on it based upon the model we just trained. Then we print the label and the prediction. The goal is to see whether the prediction is right, meaning our model is accurate.

We could also have calculated the accuracy. That is:

(number of times prediction = label) / (number of records)

We create a **QueryCursor** with **cursor.getAll**. Then we loop through it and use **getValue()** to retrieve the Vector and **get()** to retrieve individual vector elements. We use the Vectorutil **copyOfRange(start,end)** to copy the features to a features vector. The label is the first element **getValue.get(0**).

Then we run **predict()** to make a prediction, i.e., feed in the features and find the cluster to which those coordinates belong, i.e., the label.

```
val cursor = dataCache.query(new ScanQuery)
  val all = cursor.getAll


  for (i <- 0 until all.size() - 1) {
    var r = all.get(i)

    val features = r.getValue.copyOfRange(1, r.getValue.size())
    val label = r.getValue.get(0)
    printf("label=%.2f = features=", label)

    for (f <- features.asArray()) {
      printf("%.2f,", f)
```

```
    }
    print("\n")

    val prediction = mdl.predict(features)


    System.out.printf("prediction = %.2f \n", prediction.toDouble)

    println("============================================")
```

Here are the results:

```
centers 5.01 3.42 1.46 0.24

centers 5.94 2.77 4.26 1.33


============================================
label 0.00 5.10,3.50,1.40,0.20,
prediction 0.00
============================================
label 0.00 5.40,3.70,1.50,0.20,
prediction 0.00
============================================
label 0.00 5.40,3.40,1.70,0.20,
prediction 0.00
============================================
label 0.00 4.80,3.10,1.60,0.20,
prediction 0.00
============================================
label 0.00 5.00,3.50,1.30,0.30,
prediction 0.00


...


============================================
label 1.00 5.20,2.70,3.90,1.40,
prediction 1.00
============================================
label 1.00 5.60,2.50,3.90,1.10,
prediction 1.00
============================================
label 1.00 5.70,2.60,3.50,1.00,
prediction 1.00
============================================
```

```
label 1.00 5.50,2.50,4.00,1.30,
prediction 1.00
```

The complete code

Now, to run the complete code you need this Scala build.sbt file.

```
{
libraryDependencies ++= Seq(
  "org.apache.ignite" % "ignite-core" % "2.8.1",
"org.apache.ignite" % "ignite-ml" % "2.8.1" )
}
```

And here is the complete code:

```
package com.bmc.ignite


import java.util.UUID
import org.apache.ignite.Ignite

import org.apache.ignite.Ignition
import org.apache.ignite.cache.affinity.rendezvous.RendezvousAffinityFunction
import org.apache.ignite.configuration.CacheConfiguration
import org.apache.ignite.configuration.IgniteConfiguration
import org.apache.ignite.ml.clustering.kmeans.{KMeansModel, KMeansTrainer}

import org.apache.ignite.ml.math.primitives.vector.Vector
import org.apache.ignite.ml.math.primitives.vector.VectorUtils

import org.apache.ignite.ml.dataset.feature.extractor.Vectorizer
import org.apache.ignite.ml.dataset.feature.extractor.impl.DummyVectorizer

import org.apache.ignite.cache.query.ScanQuery


object Main extends App {

  val PersistencePath = "/Users/walkerrowe/Downloads/ignite"
  val WalPath = "/Users/walkerrowe/Downloads/wal"

  val config = new IgniteConfiguration()

  val ignite = Ignition.start(config)

  val dataCache = getCache(ignite)

  val file =
```

```
"/Users/walkerrowe/Documents/igniteSource/ignite/examples/src/main/resources/
datasets/two_classed_iris.csv"

  val bufferedSource = io.Source.fromFile(file)

  var i: Integer = 0
  for (line <- bufferedSource.getLines) {

    val cols: Array = line.split("\\s+").map(_.trim)

    dataCache.put(i, VectorUtils.of(cols(0).toDouble,
      cols(1).toDouble, cols(2).toDouble,
      cols(3).toDouble, cols(4).toDouble))
    i = i + 1
  }
  bufferedSource.close

  val vectorizer = new
DummyVectorizer().labeled(Vectorizer.LabelCoordinate.FIRST)

  val trainer = new KMeansTrainer()
  trainer.withAmountOfClusters(2)

  val mdl = trainer.fit(ignite, dataCache, vectorizer)

  val centers: Array = mdl.getCenters

  for (c <- centers) {
    System.out.print("centers ")
    for (a <- c.asArray()) {
      printf("%.2f ", a)
    }
    System.out.println("\n")
  }

  val cursor = dataCache.query(new ScanQuery)
  val all = cursor.getAll


  for (i <- 0 until all.size() - 1) {
    var r = all.get(i)

    val features = r.getValue.copyOfRange(1, r.getValue.size())
    val label = r.getValue.get(0)
    printf("label=%.2f = features=", label)

    for (f <- features.asArray()) {
```

```
      printf("%.2f,", f)
    }
    print("\n")

    val prediction = mdl.predict(features)


    System.out.printf("prediction = %.2f \n", prediction.toDouble)

    println("==========================================")

  }


  private def getCache(ignite: Ignite) = {
    val cacheConfiguration = new CacheConfiguration
    cacheConfiguration.setName("ML_EXAMPLE_" + UUID.randomUUID)
    cacheConfiguration.setAffinity(new RendezvousAffinityFunction(false, 10))
    ignite.createCache(cacheConfiguration)
  }

}
```