proj.scala

```
import org.apache.spark.rdd.RDD


import scala.math


val pathSeparator = File.separator


// this method computes the tf-idf score given a string to search for
and the RDD of the index


def computeScore(term: String, rddMap: RDD[(String,
(Int,Vector[String], Vector[Int]))], fileCount: Long): String = {
  val res = rddMap.lookup(term)
  if (res.length > 0) {
    val (tc, locations, df) = res(0)
    if (locations.length > 0 ) {


      // inverse documement frequency
      val idf  =
scala.math.log10(fileCount.toDouble/locations.length.toDouble)


      // weight the term freq by the idf
      val tfIdf = df.map(score=> score*idf)


      //get the index with the maximum score
      val maxIndex = tfIdf.indices.maxBy(tfIdf)
```

```
      return locations(maxIndex)
    }
    return ""
  }
  return ""
}
// we need to visit dirs recursively
sc.hadoopConfiguration.set("mapreduce.input.fileinputformat.input.dir.
recursive","true")


// I'm on a 4-core machine, so I used 8 partitions
val textFiles = sc.wholeTextFiles("hdfs://sandbox-
hdp.hortonworks.com:8020/tmp/proj/extracted/*/*", 8)
val fileCount = textFiles.count


// This is heavily adapted from just-enough-scala-for-spark tutorial
// this is the giant RDD of the inverted index
val giantRDD =textFiles.flatMap {
  case (location, contents) =>
  val words = contents.split("""\W+""").
  filter(word => word.size > 2). // only look at meaningful words
  filter(word => word.matches("[a-zA-Z]+"))
  val fileName = location
  words.map(word => ((word.toLowerCase, fileName), 1))
  }.
  reduceByKey((count1, count2) => count1 + count2).
  map {
    case ((word, fileName), count) => (word, (fileName, count))
  }. groupByKey.
  sortByKey(ascending = true).
  map {
```

```
    case (word, iterable) =>
    val vect = iterable.toVector.sortBy {
    case (fileName, count) => (-count, fileName)
  }
  val (locations, counts) = vect.unzip
  val totalCount = counts.reduceLeft((n1,n2) => n1+n2)
  (word, (totalCount, locations, counts))
}
giantRDD.persist
```

User.scala

```
import java.io.File
import org.apache.spark.rdd.RDD
import scala.math
import scala.util.matching.Regex
val pathSeparator = File.separator


// this gets the matching document from the meta document
def getMatchingDocument(term: String, document: String) {

  // gymnastics to split the file into html docs
  val wikiFiles = sc.wholeTextFiles(document, 4)
  val wikiRDD =wikiFiles.flatMap {
  case (location, contents) =>
  val docs = contents.split("""url=""")
  docs.filter(doc =>
doc.split("""title=""")(0).replaceAll("\\s+","").contains("https")).
```

```
  map(doc =>
(doc.split("""title=""")(0).replaceAll("\\s+","").replaceAll("\"",""),
(doc)))
  }.
  sortByKey(ascending = true).
  flatMap {
  case (location, contents) =>
  val words =  contents.split("""\W+""").
  filter(word => word.size > 2).
  filter(word => word.matches("[a-zA-Z]+"))
  words.map(word => ((word.toLowerCase, location), 1))
  }.
  reduceByKey((count1, count2) => count1 + count2).
  map {
    case ((word, fileName), count) => (word, (fileName, count))
  }.
  groupByKey.
  sortByKey(ascending = true).
  map {
    case (word, iterable) =>
    val vect = iterable.toVector.sortBy {
    case (fileName, count) => (-count, fileName)
  }
    val (locations, counts) = vect.unzip
    val totalCount = counts.reduceLeft((n1,n2) => n1+n2)
    (word, (totalCount, locations, counts))
  }
  val finalDoc = computeScore(term, wikiRDD, wikiRDD.count)
  if (finalDoc.length >0) {
    println (s"""Matching Document according to tf-idf score is
$finalDoc""")
```

```
  } else {
    println (s"""No matching document found""")
  }
}


// Prompt the user for input
Console.println("Hello and welcome to the google-killer :)")
val name = readLine("Enter Search Term: ")
val doc = computeScore(name, giantRDD, fileCount)
if (doc.length >0) {
  println (s"""Matching Document according to tf-idf score is $doc""")
  getMatchingDocument(name, doc)
} else {
  println (s"""No matching document found""")
}
```