1. Go to spark installation folder and then create a folder for your code. I compiled and ran my code using SBT and vi editor. Create .sbt file to tell sbt which version of scala to execute and its library dependencies.
   Create a folder structure source/main/scala. Place your scala code here.

   To compile give :sbt package
   To run: /Users/Alya/Documents/spark-1.2.1-bin-hadoop2.4/bin/spark-submit --class "ImageReading" --master local target/scala-2.10/simple-project_2.10-1.0.jar

Jar file is in target/target/scala-2.10

Answers to Questions

1. When run my program with k=10 and single core using
   --master local
   Runtime=453993 milli seconds(454 Sec)

   When run with
   --master local[*]
   Runtime = 264449 milli seconds(264 Sec)

I see that the run time was significantly improved when all the cores were used. My machine has 8 cores.
As the number of cores increases, run time decreases. If multiple cores are across multiple machines, then I expect the run time improves significantly. Because work is distributed among multiple machines and is done in parallel manner.

2. The files for different principal components are in matrixdata folder.
When k=10 Frobenius Norm is 240.27003780805987

When k=50 Frobenius Norm is 151.25964587472632

When k=100 Frobenius Norm is 97.86995092022642

When k=150 Frobenius Norm is 66.48373708335372

when k=200 Frobenius Norm is 45.296879542822865

when k=250 Frobenius Norm is 29.40466926740502

when k=300 Frobenius Norm is 16.982321668268565

I see that as k increases the reconstruction error decreases. This is because the more components we get rid of, the error of the image increases. Based on rule of thumb keeping 80 - 90% of energy, I think choosing some value between 200 to 250 might be ideal choice of k, because the error is from 45 to 29(80% 256=51 and 90% 256=25.6)

3.

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import java.io.File
import java.awt.image.BufferedImage
import javax.imageio.ImageIO
import breeze.linalg.svd.SVD
import breeze.linalg.qr.QR
import breeze.linalg.qrp.QRP
import breeze.linalg._
import breeze.linalg.svd.SVD
import scala.collection.mutable.ArrayBuffer
import java.util.Arrays

object newSpan {

def main(args: Array[String]){
        val conf = new SparkConf().setAppName("Span Application")
        val sc = new SparkContext(conf)
        var imgFiles = new ArrayBuffer[String]()

for(f<-0 until 1) {
if(f<=9)
        imgFiles+=
"/Users/Alya/Documents/spark-1.2.1-bin-hadoop2.4/sparkExperiments/SVDTest/images/000"
+f+".png"
else if(f>9 && f<=99)
        imgFiles+=
"/Users/Alya/Documents/spark-1.2.1-bin-hadoop2.4/sparkExperiments/SVDTest/images/00"+f
+".png"
else if(f>100 && f<=999)
        imgFiles+=
"/Users/Alya/Documents/spark-1.2.1-bin-hadoop2.4/sparkExperiments/SVDTest/images/0"+f+
".png"
}

var fparallel = sc.parallelize(imgFiles)
```

```
def flatFunc(imgName:String):DenseVector[Double] ={
val img = ImageIO.read(new File(imgName))
val rows = img.getHeight()
val cols = img.getWidth()
val imageMatrix = breeze.linalg.DenseMatrix.zeros[Double](420,420)

for ( i <- 0 until cols; j <- 0 until rows ) {
      imageMatrix(i,j) =(img.getRGB(j,i) & 0x000000ff).toDouble
      }
var flatVec = breeze.linalg.DenseVector.zeros[Double](1000)
flatVec = imageMatrix.toDenseVector(0 to imageMatrix.rows*imageMatrix.cols-1)
flatVec
}

def assignFunc(flattedVector:DenseVector[Double]): collection.mutable.Map[Int,Double]={
var assignMap = collection.mutable.Map[Int,Double]()
for(i <-0 until flattedVector.length)
assignMap+=i->flattedVector(i)
assignMap
}

var max = 0.0
var min = 255.0

def spanFunc(key:Int, data: Iterable[Double],key:Int, data: Iterable[Double]): Double ={
var span = 0.0
for(eachData <- data){
if(min>eachData) min = eachData
if(max<eachData) max = eachdata
}
span = data.max-data.min
span
}


var flatVecData =
fparallel.map(fname=>flatFunc(fname)).map(vect=>assignFunc(vect)).reduceByKey(spanFun
c)

for(vec<-flatVecData)
println(vec)
```

```
//We already have span for all 17600 dimensions in flatVecData. I used rankMap to store rank of each dimension as key and
dimension with  maximum value as value. rankMap(rank, dimension with max span)

var rankMap = collection.mutable.Map[Int,Int]()

for(i<-0 until flatVecData.length){
rankMap += i -> (flatVecData.indexOf(flatVecData.max))
flatVecData(flatVecData.indexOf(flatVecData.max)) = 0
}

//This picks the dimensions of 10 pixels with largest span.
for(i <- 0 until 10)
println("rank "+i+" is given to "+ rankMap(i) +" dimension.")*/
}
}
```

I parallelized the images and flattened the matrix row wise. Collected the results in the form of array of dense vectors. Passed these values to a map function for assigning index as key and value as the pixel value. Using reduceByKey span for each dimension is calculated by taking difference between minimum and maximum value. Stored it in a map with key as rank and value as dimension index with maximum span.
Also it lists the indices of 10 pixels with largest span.

4.

Calculating Hyperplane

I have spans of each dimension. Then I rank the dimensions according to the span. I calculate the probability that each span is chosen by taking span_of_each_Dimension/ total_span_of_allDimentions. Now that I have probabilities, I represent these fractional values in the form of an array of size= total_span(Denominator of probability), the values of the array as the index of each span. The span for a dimension represents the number of array values to be filled with respective dimension.
Thus when get_hyperplane(i) is called, this array gives the index of that particular dimension.

Calculating Threshold

Between the minimum and maximum span, let us chose 20 bins and see how many dimensions fall with in range of [min[i]+j×span[i]/20, min[i]+ (j + 1) × span[i]/20]. Then minimum value among the range is chosen and lets denote by s. Then tresh hold for that dimension is set to Dim[i] = min[i] + s × span[i]/20.

Now we have hyperplane and treshold for each dimension.
We can calculate hash value for an image by looping through all the 176400 dimensions calculating p value.

```
String Sig ="";
for i = 1;i ≤ M;i + + do
        Threshold = get threshold(i) ;
        Hyperplane = get hyperplane(i) ;
        if inputVector[Hyperplane] ≤ Threshold then
                p =1
        else
                 p =0;
        Convert p to String and add to the tail of Sig ;
```

Instead of having 17600 Signatures we have only p valued signatures.

We have p signatures for all 1000 images. Now we can hash them into buckets by calculating Hamming distance between the pairs of values. It is similar to performing XOR operation between p signatures. If the hamming distance between pairs of images is 0 then they are grouped in to same bucket. Otherwise they are in different buckets.

Now the similarity between the 2 matrices are found using

```
Length = getLength(indexList)
        for i = 1;i ≤ Length;i + + do
        for j = 1; j ≤ Length; j + + do
        if i = j then
        subSimM at[i, j] = simF unc(i, j) ;
        else
         subSimM at[i, j]=0;
Output to File(subSimM at) ;
```