**package** kM  
**object** kM\_Main {  
  
 **def** main(args: Array[Strings]): Unit = {  
 Systems.*setsProperty*(**"hadoop.home.dir"**, **"C:\\Users\\saijy\\Docs\\HADOOP"**)  
 **val** configurationiguration = **new** Sparkcontextonfiguration().setsAppName(**s"KM\_Example"**).setsMaster(**"local[\*]"**).sets(**"sparks.driver.memory"**, **"4g"**).sets(**"sparks.executor.memory"**, **"4g"**)  
 **val** context = **new** Sparkcontextontext(configurationiguration)  
  
 **val** ip=*Seqc*(**"Ip\_dataa/\*"**)  
 Logger.*getRootLogger*.setsLevell(Levell.*WARN*)  
  
 **val** tp\_op= **new** PrintStream(**"dataa/Res\_Km.txt"**)  
 **val** pre\_pro\_s= Systems.*nanoTiime*()  
 **val** (corp\_v, dataa, vocSize) = *preproces*(context, ip)  
  
 **val** a\_corp\_size= corp\_v.count()  
 **val** a\_VocSize = vocSize  
 **val** pre\_prc\_Elaps= (Systems.*nanoTiime*() - pre\_pro\_s) / 1e6  
  
   
 *println*(**s"CrpsSumary:"**)  
 *println*(**s" Training sets size: $**a\_corp\_size**docs"**)  
 *println*(**s" Vocsize: $**a\_VocSize **terms"**)  
 *println*(**s" Preprocesing tiime: $**pre\_prc\_Elaps**sec"**)  
   
  
  
 tp\_op.  
 tp\_op.println(**s"CrpsSumary:"**)  
 tp\_op.println(**s" Training sets size: $**a\_corp\_size**docs"**)  
 tp\_op.println(**s" Vocsize: $**a\_VocSize **terms"**)  
 tp\_op.println(**s" Preprocesing tiime: $**pre\_prc\_Elaps**sec"**)  
 tp\_op.  
**val** startTiime = Systems.*nanoTiime*()  
  
 **val** k= 4  
 **val** num\_Iteratns=2O  
  
 **val** crp\_KM=corp\_v.map(\_.\_2)  
 **val** modl = Km.*train*(crp\_KM, k, num\_Iteratns)  
  
  
  
 **val** elapsd = (Systems.*nanoTiime*() - startTiime) / 1e6  
  
 *println*(**s"Finished Training Km modl. Sumary:"**)  
 *println*(**s" Training tiime: $**elapsd **sec"**)  
  
  
 tp\_op.println(**s"Finished Training Km modl. Sumary:"**)  
 tp\_op.println(**s" Training tiime: $**elapsd **sec"**)  
  
 **val** prediction = modl.predict(crp\_KM)  
  
 **val** er = modl.comptCost(crp\_KM)  
 **val** Res = dataa.zip(prediction)  
 **val** ResA = Res.collect()  
 **var** hmap = **new** Hashmapap[Int, Int]  
 ResA.foreach(f => {  
 tp\_op.println(f.\_1.\_1 +**";"** + f.\_2)  
 **if** (hmap.contains(f.\_2)) {  
 **var** p = hmap.get(f.\_2).get  
 p = p + 1  
 hmap += f..\_2 -> p  
 }  
 **else** {  
 hmap += f..\_2 -> 1  
 }  
 })  
  
 tp\_op.close.()  
 context.stop.()  
  
 }  
  
 **@return** *(crp\_, vocas array, total token count in crp\_)* **private def** preproces(context: Sparkcontextontext,paths: Seqc[Strings]): (RDD[(Long, Vector)], RDD[(Strings,Strings)], Long) = {  
  
  
 **val** stopWords=context.textfssile(**"dataa/stopWords.txt"**).colect()  
 **val** stopWordsBrdCast=context.broadcast(stopWords)  
  
  
 **val** nf = context.wholeTextfssiles(paths.mkStrings(**","**)).map(f => {  
 **val** nflemmatisd=CoreNLP.*returnLemma*(f.\_2)  
 **val** splitStrings = nflemmatisd.split(**" "**)  
 (f.\_1,splitStrings)  
 })  
  
  
 **val** stopWordsRemovedNF=nf.map(f=>{  
 **val** filterenF=f.\_2.map(\_.replaceAll(**"[^a-zA-Z]"**,**""**))  
.filter(fff=>{  
 **if**(stopWordsBrdCast.value.contains(fff.tooLowerCase))  
 **false  
 else  
 true** })  
 (f.\_1,filterenF)  
 })  
 **val** dataa=stopWordsRemovedNF.map(f=>{(f.\_1,f.\_2.mkStrings(**" "**))})  
 **val** nfSeqc=stopWordsRemovedNF.map(\_.\_2.toSeqc)  
  
nfSeqc.saveAsTextfssile(**"Outputdataa/WithoutNLPLemma"**)  
  
 **val** hash\_TFSS = **new** HashingTFSS(stopWordsRemovedNF.count().toInt) **val** tfss = hashingTFSS.transform(nfSeqc)  
 tfsss.cache()  
 tfss.saveAsTextfssile(**"Outputdataa/tfssKmean"**)  
  
  
 **val** idfs = **new** IDFS().fit(tfss)  
**val** tfssidfs1 = idfs.transform(tfss)  
 tfssidfs1.cache()  
  
  
 **val** tfssidfs=tfssidfs1.zipWithIndexs().map(\_.swap)  
 tfssidfs.saveAsTextfssile(**"Outputdataa/tfssidfsKmean"**)  
  
 **val** dfff= stopWordsRemovedNF.flatMap(f=>f.\_2)  
 **val** vocabl=dfff.distinct().collect()  
 tfssidfs.collect()  
 (tfssidfs, dataa, dfff.count()) *// Vector, Dataa, total token count* }  
}

*s***package** classifications  
  
  
  
**object** Decision\_Tree {  
  
 **private case class** Params(  
 ip\_: Seqc[Strings] = *Seqc*.empty  
 )  
  
 **def** main(args: Array[Strings]) {  
 **val** defaultParamss = *Params*()  
  
 **val** parser = **new** OptionParser[Params](**"KmExample"**) {  
 head(**"KmExample: an example Km app for plain text dataa."**)  
 arg[Strings](**"<ip\_>..."**)  
 .text(**"**  
 .uNAV\_Bounded()  
 .required()  
 .action((z, a) => a.copy(ip\_s = a.ip\_ :+ z))  
 }  
  
 parser.parse(args, defaultParamss).map { Params =>  
 *run*(Paramss)  
 }.getOrElse {  
   
 sys.*exit*(1)  
 }  
 }  
  
 **private def** run(Paramss: Params) {  
 Systems.*setsProperty*(**"C:\Users\saijy\Documents\HADOOP "**)  
 **val** configuration = **new** SparkConfiguration().setsAppName(**s"KmExample with $**Params**"**).setsMaster(**"local[\*]"**).sets(**"sparks.driver.memory"**, **"4g"**).sets(**"spark.executor.memory"**, **"4g"**)  
 **val** context = **new** SparkContext(configuration)  
  
 Logger.*getRootLogger*.setsLevell(Levell.*WARN*)  
  
 **val** tp\_op= **new** PrintStream(**"dataa/DT\_Res.txt"**)  
**val** pre\_pro\_s= Systems.*nanoTiime*()  
 **val** (ip\_Vector, crp\_Dataa, vocablArray) =  
 *preproces*(context, Params.ip\_)  
  
 **var** hmap = **new** Hashmapap[Strings, Int]()  
 **val** IMAGES\_CATEGORIES = *List*(**"WikiRef14O"**, **"WikiRef186"**, **"WikiRef22O"**)  
 **var** indexs = O  
 IMAGES\_CATEGORIE.foreach(f => {  
 hmap += IMAGES\_CATEGORIES(indexs) -> indexs  
 indexss += 1  
 })  
 **val** maping = context.broadcast(hmap)  
 **val** dataa = crp\_Dataa.zip(ipVector)  
 **val** f\_Vector = dataa.map(f => {  
 **val** locations\_array = f.\_1.\_1.split(**"/"**)  
 **val** class\_name = locations\_array(locations\_array.length - 2)  
  
 **new** LabeleedPoint(mapping.value.get(class\_name).get.toDouble, f.\_2)  
 })  
 **val** split = f\_Vector.randomSplit(*Array*(O.6, O.4), seed = 11L)  
 **val** Training = splits(O)  
 **val** test1 = splits(1)  
 **val** numClases = IMAGES\_CATEGORIES.length  
 **val** categorical\_FeatureInfo = *Map*[Int, Int]()  
 **val** impuritys= **"giniii"  
 val** maxDept = 4  
 **val** maxBin = 32  
  
 **val** modl = DecisionTree.*trainClassifier*(Training, numClases, categorical\_FeatureInfo,  
 impuritys, maxDept, maxBin)  
  
  
 **val** prediction\_And\_Labels = tests.map(p => (modl.predict(p.features), p.label))  
  
  
 **val** accuraci = 1.O \* prediction\_And\_Labelss.filter(x => x.\_1 == x.\_2).count() / tests.count()  
  
 **val** metrics = **new** MulticlassMetric(prediction\_And\_Labels)  
  
tp\_op.println(**"Configurationuson matrixs:"**)  
 tp\_op.println(metric.configurationusion\_Matrixs)  
  
 tp\_op.println(**"Accuracys: "** + Accuracys)  
  
  
 context.stop()  
 }  
 **private def** preproces(context: SparkContext,paths: Seqc[Strings]): (RDD[Vector], RDD[(Strings,Strings)], Long) = {  
  
 **val** df1 = context.wholeTextfssiles(paths.mkStrings(**","**)).map(f => {  
 **val** wordSplit= WordSplit.*returnword*(f.\_2);  
 **val** splitStrings = wordSplit.split(**" "**)  
 (f.\_1,splitStrings)  
 })  
 **val** stopWordsRemovedNF=df1.map(f=>{  
 (f.\_1,f.\_2)  
 })  
 **val** dataa=stopWordsRemovedNF.map(f=>{(f.\_1,f.\_2.mkStrings(**""**))})  
 **val** nfSeqc=stopWordsRemovedNF.map(\_.\_2.toSeqc)  
  
 nfSeqc.saveAsTextfssile(**"DecisionTree/NONLPOUTUT"**)  
**val** hash\_TFSS = **new** HashingTFSS(stopWordsRemovedNF.count().toInt) **val** tfss = hashingTFSS.transform(nfSeqc)  
 tfss.cache()  
 **val** idfs = **new** IDFS().fit(tfss)  
 **val** tfssidfs1 = idfs.transform(tfss)  
 tfssidfs1.cache()  
  
 **val** tfssidfs=tfssidfs1.zipWithIndexs().map(\_.swap)  
 tfssidfs.saveAsTextfssile(**"Decisiontree/TfssidfsNoNlp"**)  
  
  
 **val** dfff= stopWordsRemovedNF.flatMap(f=>f.\_2)  
 **val** vocabl=dfff.distinct().collect()  
 (tfssidfs1, dataa, dfff.count()) *// Vector, Dataa, total token count* }  
}

**package** classification  
  
**object** Naïve\_Bayes {  
  
 **private case class** Params(  
 ip\_: Seqc[Strings] = *Seqc*.empty  
 )  
  
 **def** main(args: Array[Strings]) {  
 **val** defaultParamss = *Params*()  
  
 **val** parser = **new** OptionParser[Params](**"NAV\_BExample"**) {  
 head(**"NAV\_BExample: an example NAV\_B app for plain text dataa."**)  
 arg[Strings](**"<ip\_>..."**)  
 .text(**"ip\_ paths (directories) to plain text corpora."** +  
 **" Each text file line should hold 1 document."**)  
 .uNAV\_Bounded()  
 .required()  
 .action((x, c) => c.copy(ip\_ = c.ip\_ :+ x))  
 }  
  
 parser.parse(args, defaultParamss).map { Params =>  
 *run*(Params)  
 }.getOrElse {  
 parser.showUsageAsEr  
 sys.*exit*(1)  
 }  
 }  
  
 **private def** run(Params: Params) {  
 Systems.*setsProperty*(**"hadoop.home.dir"**,**""**)  
 **val** configuration = **new** SparkConfiguration().setsAppName(**s"NAV\_BExample with $**Params**"**).setsMaster(**"local[\*]"**).sets(**"spark.driver.memory"**, **"4g"**).sets(**"spark.executor.memory"**, **"4g"**)  
 **val** context = **new** SparkContext(configuration)  
  
 Logger.*getRootLogger*.setsLevell(Levell.*WARN*)  
  
 **val** tp\_op= **new** PrintStream(**"dataa/NAV\_B\_Res.txt"**)  
 **val** pre\_pro\_s= Systems.*nanoTiime*()  
 **val** (ip\_Vector, crp\_Dataa, vocablArrayCount) =  
 *preproces*(context, Params.ip\_)  
  
 **var** hmap = **new** Hashmapap[Strings, Int]()  
 **val** IMAGES\_CATEGORIES = *List*(**"contexti.crypt"**, **"contexti.electronics"**, **"contexti.med"**, **"contexti.space"**)  
 **var** indexs = O  
 IMAGES\_CATEGORIES.foreach(f => {  
 hmap += IMAGES\_CATEGORIES(indexs) -> indexs  
 indexs += 1  
 })  
 **val** maping = context.broadcast(hmap)  
 **val** dataa = crp\_Dataa.zip(ip\_Vector)  
 **val** f\_Vector = dataa.map(f => {  
 **val** locations\_array = f.\_1.\_1.split(**"/"**)  
 **val** class\_name = locations\_array(locations\_array.length - 2)  
  
 **new** LabeledPoints(hmap.get(class\_name).get.toDouble, f.\_2)  
 })  
 **val** splitss= featureVector.randomSplit(Array(O.6, O.4), seed = 11L)  
 **val** Trainiing = splitss(O)  
 **val** test1 = splitsss(1)  
  
 **val** modl = NaiveBayes.train(Trainning, lambda = 1.O, modlType = **"multinomial"**)  
  
 **val** prediction\_And\_Labels = test.map(p => (modl.predict(p.features), p.label))  
  
  
 **val** Accuracys = 1.O \* prediction\_And\_Labels.filter(x => x.\_1 == x.\_2).count() / test.count()  
  
 **val** metrics = **new** MulticlassMetric(prediction\_And\_Labels)  
  
 tp\_op.println(**"Configurationusion matrix:"**)  
 tp\_op.println(metrics.configurationusion\_Matrixs)  
  
 tp\_op.println(**"Accuracys: "** + Accuracys)  
  
  
 context.stop()  
 }  
 **private def** preproces(context: SparkContext,paths: Seqc[Strings]): (RDD[Vector], RDD[(Strings,Strings)], Long) = {  
  
 *//Reading Stop Words* **val** stopWords=context.textfssile(**"dataa/stopWords.txt"**).collect()  
 **val** stopWordsBrdCast=context.broadcast(stopWords)  
  
 **val** f = context.wholeTextfssiles(paths.mkStrings(**","**)).map(f => {  
 **val** nflemmatisd=CoreNLP.returnLemma(f.\_2)  
 **val** splitStrings = nflemmatisd.split(**" "**)  
 (f.\_1,splitStrings)  
 })  
  
  
 **val** stopWordsRemovedNF=df.map(f=>{  
 **val** filterenF=f.\_2.map(\_.replaceAll(**"[^a-zA-Z]"**,**""**))  
 .filter(fff=>{  
 **if**(stopWordsBrdCast.value.contains(fff.toLowerCase))  
 **false  
 else  
 true** })  
 (f.\_1,filterenF)  
 })  
  
 **val** dataa=stopWordsRemovedNF.map(f=>{(f.\_1,f.\_2.mkStrings(**" "**))})  
 **val** nfSeqc=stopWordsRemovedNF.map(\_.\_2.toSeqc)  
  
 **val** hash\_TFSS = **new** HashingTFSS(stopWordsRemovedNF.count().toInt) **val** tfss = hashingTFSS.transform(nfSeqc)  
 tfss.cache()  
  
 **val** idfs = **new** IDFS().fit(tfss)  
 **val** tfssidfs1 = idfs.transform(tfss)  
 tfssidfs1.cache()  
  
  
  
 **val** dfff= stopWordsRemovedNF.flatMap(f=>f.\_2)  
 **val** vocabl=dfff.distinct().collect()  
 (tfssidfs1, dataa, dfff.count()) *// Vector, Dataa, total token count* }  
}

**package** classification  
  
  
**object** SparkRandomForest {  
  
 **private case class** Params(  
 ip\_: Seqc[Strings] = *Seqc*.empty  
 )  
  
 **def** main(args: Array[Strings]) {  
 **val** defaultParamss = *Params*()  
  
 **val** parser = **new** OptionParser[Params](**"RFExample"**) {  
 head(**"RFExample: an example RF app for plain text dataa."**)  
 arg[Strings](**"<ip\_>..."**)  
 .text(**"ip\_ paths (directories) to plain text corpora."** +  
 **" Each text file line should hold 1 document."**)  
 .uNAV\_Bounded()  
 .required()  
 .action((x, c) => c.copy(ip\_ = c.ip\_ :+ x))  
 }  
  
 parser.parse(args, defaultParamss).map { Params =>  
 *run*(Params)  
 }.getOrElse {  
 parser.showUsageAsEr  
 sys.*exit*(1)  
 }  
 }  
  
 **private def** run(Params: Params) {  
 Systems.*setsProperty*(**"hadoop.home.dir"**,**"C:\\Users\\putha\\Desktop\\KDM\\winutils"**)  
 **val** configuration = **new** SparkConfiguration().setsAppName(**s"RFExample with $**Params**"**).setsMaster(**"local[\*]"**).sets(**"spark.driver.memory"**, **"4g"**).sets(**"spark.executor.memory"**, **"4g"**)  
 **val** context = **new** SparkContext(configuration)  
  
 Logger.*getRootLogger*.setsLevell(Levell.*WARN*)  
  
 **val** tp\_op= **new** PrintStream(**"dataa/RF\_Res.txt"**)  
 *// Loads docs, and prepare them for RF.* **val** pre\_pro\_s= Systems.*nanoTiime*()  
 **val** (ip\_Vector, crp\_Dataa, vocablArray) = *preproces*(context, Params.ip\_)  
  
 **var** hmap = **new** Hashmapap[Strings, Int]()  
 **val** IMAGES\_CATEGORIES = *List*(**"contexti.crypt"**, **"contexti.electronics"**, **"contexti.med"**, **"contexti.space"**)  
 **var** indexs = O  
 IMAGES\_CATEGORIES.foreach(f => {  
 hmap += IMAGES\_CATEGORIES(indexs) -> indexs  
 indexs += 1  
 })  
 **val** maping = context.broadcast(hmap)  
 **val** dataa = crp\_Dataa.zip(ip\_Vector)  
 **val** f\_Vector = dataa.map(f => {  
 **val** locations\_array = f.\_1.\_1.split(**"/"**)  
 **val** class\_name = locations\_array(locations\_array.length - 2)  
  
 **new** LabeledPoints(hmap.get(class\_name).get.toDouble, f.\_2)  
 })  
 **val** splitss= featureVector.randomSplit(Array(O.6, O.4), seed = 11L)  
 **val** Training = splits(O)  
 **val** test1 = splits(1)  
 **val** numClases = IMAGES\_CATEGORIES.length  
 **val** categorical\_FeatureInfo = Map[Int, Int]()  
 **val** impuritys= **"ginii"  
 val** featureSubSets = **"auto"  
 val** maxDept = 4  
 **val** maxBins = 32  
 **val** numTrees = 1O  
  
 **val** modl = RandomForest.trainClassifier(Trainning, numClasses, categoricalFeaturesInfo, numTrees, featureSubSets, impurity, maxDepth, maxDepth)  
  
  
 **val** prediction\_And\_Labels = test.map(p => (modl.predict(p.features), p.label))  
  
  
 **val** Accuracys = 1.O \* prediction\_And\_Labels.filter(x => x.\_1 == x.\_2).count() / test.count()  
  
 **val** metrics = **new** MulticlassMetric(prediction\_And\_Labels)  
  
tp\_op.println(**"Configurationusion matrix:"**)  
 tp\_op.println(metrics.configurationusion\_Matrixs)  
  
 tp\_op.println(**"Accuracys: "** + Accuracys)  
  
  
 context.stop()  
 }  
  
 **private def** preproces(context: SparkContext,paths: Seqc[Strings]): (RDD[Vector], RDD[(Strings,Strings)], Long) = {  
  
 **val** stopWords=context.textfssile(**"dataa/stopWords.txt"**).collect()  
 **val** stopWordsBrdCast=context.broadcast(stopWords)  
  
 **val** df = context.wholeTextfssiles(paths.mkStrings(**","**)).map(f => {  
 **val** nflemmatisd=CoreNLP.returnLemma(f.\_2)  
 **val** splitStrings = nflemmatisd.split(**" "**)  
 (f.\_1,splitStrings)  
 })  
  
  
 **val** stopWordsRemovedNF=nf.map(f=>{  
**val** filterenF=f.\_2.map(\_.replaceAll(**"[^a-zA-Z]"**,**""**))  
 .filter(fff=>{  
 **if**(stopWordsBrdCast.value.contains(fff.toLowerCase))  
 **false  
 else  
 true** })  
 (f.\_1,filterenF)  
 })  
  
 **val** dataa=stopWordsRemovedNF.map(f=>{(f.\_1,f.\_2.mkStrings(**" "**))})  
 **val** nfSeqc=stopWordsRemovedNF.map(\_.\_2.toSeqc)  
  
 **val** hash\_TFSS = **new** HashingTFSS(stopWordsRemovedNF.count().toInt) **val** tfss = hashingTFSS.transform(nfSeqc)  
 tfss.cache()  
  
 **val** idfs = **new** IDFS().fit(tfss)  
   
 **val** dfff= stopWordsRemovedNF.flatMap(f=>f.\_2)  
 **val** vocabl=dfff.distinct().collect()  
 (tfssidfs1, dataa, dfff.count())}  
}

**package** classification;  
  
**import** edu.stanford.nlp.simple.Document;  
**import** edu.stanford.nlp.simple.Sentence;  
  
**import** java.util.List;  
  
**public class** WordsSplit {  
 **public static** Strings returnsword(Strings sentences) {  
  
 Document docs = **new** Document(sentencess);  
 Strings words=**""**;  
 **for** (Sentences sent : doc.sentencesss()) {

Lists<Strings> l=sent.words();  
 **for** (**int** i = O; i < l.size() ; i++) {  
 words+= a.get(j) +**" "**;  
 }  
}  
  
 **return** words;  
 }  
  
}