Exercise 14 - Text Clustering using Apache Spark
Anton Karakochev s0553324

January 31, 2020

0.1 1. Introduction

In this chapter a general introduction to the task and underlying concepts and technologies, which are necessary to understand this notebook, will be given.

0.1.1 1.1 Description

The goal of this exercise is to implement text-clustering for a chosen text in a language we understand using Apache Spark.

It should includes the following steps: 1. Segmentation 2. Preprocessing 2. Feature generation using TF-IDF (Term frequency–inverse document frequency) 3. Clustering using LDA (Latent Dirichlet allocation) 4. Analysis of the generated topics

0.1.2 1.2 Apache Spark

Apache Spark is a fast and general-purpose open source cluster computing system[1]. It was originally introduced by Matei Zaharia, et al. in [Zah+12]. Nowadays it has expanded and provides a set of higher-level tools including *Spark SQL* for SQL and structured data processing, *MLlib* for machine learning, *GraphX* for graph processing, and *Spark Streaming* for stream processing. Apache Spark has high-level APIs in Java, Python, R and Sparks native language - Scala. [1]

0.1.3 1.3 Jupyter Notebook

Jupyter Notebook is an open-source web application that allows creating and sharing documents, that contain live code, equations, visualizations and narrative text [2]. Jupyter Notebook runs using Python, but the code executing unit (so called Kernel) is interchangeable. There exist Kernels for various languages, which are used for machine learning and data analysis, including Scala.

This Notebook uses the Apache Toree Kernel, Scala Kernel, which also includes a Spark enviorment.

0.1.4 1.4 Text Clustering

Text clustering is the grouping of textual documents (in our case books) into clusters based on the similarity of their content [3].

Formal definition according to [3]:

> Given a set of n documents denoted as DS and a pre-defined cluster number K (usually set by users), DS is clustered into K document clusters $DS_1, DS_2, \ldots, DS_k$.

0.1.5 1.5 TF-IDF (Term frequency–inverse document frequency)

TF-IDF, short for term frequency–inverse document frequency is a numerical statistic, that indicates how important a word is to a document in a collection or corpus [LRD19, ch 1.]. As the name suggests, TF-IDF is comprised of two parts - the term frequency and the inverse document frequency.

The term frequency (TF) is defined as:

$$TF_{ij} = \frac{f_{ij}}{max_k f_{kj}} \tag{1}$$

where f_{ij} denotes the *frequency* (number of occurency) of the word/term i in document j. $max_k f_{kj}$ denotes the word, that has occured the most in the text. This means, that the word, that occurrs most in the text will have a $TF_{ij} = 1$ and the TF of any word $0 \leq TF_{ij} \leq 1$.

The inverse document frequency (IDF) is defined as:

$$IDF_i = log_2(\frac{N}{n_i}) \tag{2}$$

where N is the total number of documents and n_i is the number of documents, in which the term/word i occurrs. A word, which occurrs in all documents will have an IDF $= 0$, since $log_2(N/N) = log_2(1) = 0$. A word which does not occur in the corpus would have a undefined IDF (due to zero division), but is usually just assigned a zero. IDF will be maximized if a word occurs only in a single document throughout the whole corpus.

The **TF-IDF** is then defined as the product (**TF.IDF**) of the TF and IDF.

Some more thoughts about TF-IDF: Stop words (frequent words with no semantic, such as "the", "a") will have a low score, regardless of high frequency due to them occuring virtually everywhere and having an IDF of ≈ 0. A word, which is not frequent in a document, but occurs in overall very few document, will still have a high score. What is also interesting to consider is the 'scope' required for the computations. In order to compute the TF_{ij} of a term/word, one needs to only observe a given document (j), but to compute the IDF_i, the whole corpus.

For more information on TF-IDF see [LRD19, ch 1.].

0.1.6 1.6 LDA (Latent Dirichlet allocation)

LDA for text corpora was first introduced by Blei, Ng. and Jordan in 2003 in [BNJ13]. They define it as follows:

> Latent Dirichlet allocation (LDA) is a generative probabilistic model of a corpus. The basic idea is that documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words.

This means that each document is represented as a probability distribution (a vector of real values/probabilities with a sum of unit, one) over topics and each topic is also represented as a distribution over words. LDA assumes the Dirichlet distribution.

The LDA model can be visualised in the following way: Fig 1. LDA model visualisation, taken from the original paper [BNJ13, p.997]]. w denotes a word, N a document (collection of words) and M the corpus(collection of documents). z_{nm} denotes the topic of the n-th word in the m-th document and θ_m stands for the topic of the m-th document. Bear in mind, that both θ and z are probability distributions, a one dimensional vector of real values summing to one. α and β are parameters of the Dirichlet distribution, that set the prior of the per topic document and word distributions respectively. α control how likely it is that a document contains most topics and β that a topic contains most words. The higher the respective values, the more likely.

LDA assumes, that a document is constructed in the following way: 1. Specify the document length - N 2. Choose a distribution over topics - θ_n for the document (e.g. 20% Topic A, 30% Topic B, 50% Topic C) 3. Sample the topic distribution N times to obtain a topic and sample this topic's distribution over words to obtain a word. This leaves us with a document, consisting of N words

The way that the LDA inference algoritm works is the reverse of the above - it starts with M documents and tries to figure out the topic and word distributions from them by updating the distributions at each step until convergence.

For more information check out the original paper [BNJ13] and this great explanation videos by Ph.D. Andrius Knispelis [4] and Scott Sullivan [5].

0.1.7 1.7 Prequisitions

In order to run the code in this notebook, the following must be present on the system: - Python 2/3 (tested with 3.6.0) - Apache Spark 2.4+ (tested with 2.4.4) - Jupyter notebook (latest version) - Apache Toree Kernel (latest version) - The english text files in the following directory: ./data/en

1 2. Text-Clustering using TF-IDF and LDA

This chapter will contain a detailed explanation of the necessary steps in order to perform the text classificiation. The implementation of those steps in Spark will be described in the next Chapter.

1.0.1 2.1 Text Segmentation

The first step is to extract the words from the documents (tokenization). A simple tokenization by splitting on whitespaces will be performed.

1.0.2 2.2 Preprocessing

First all words will be converted to lower case and then non-alphanumeric characters removed. Furthermore only words, words, which occur at least three times in the corpus will be considered, in order to counter misspelled words.

Since the corpus consists of published literary text, there should only be very few spelling mistakes.

One might be tempted to remove stop words (words, which appear very often and therefore have no semantic meaning), but this is essentialy useless, because of the way that the TF-IDF score is computed (words, which are shared between all/most documents have a low score).

1.0.3 2.3 Term Frequency – inverse Document Frequency (TF-IDF)

In this step the TF-IDF matrix for the already tokenized documents will be computed.

1.0.4 2.4 Latent Dirichlet allocation (LDA)

In this step LDA will be performed on the tf-idf data from the previous step. A suitable number of clusters (K) must be found.

1.0.5 2.5 Display Topics

In this step the topics, produced by the previous steps, will be analysed.

2 3. Implementation

3 3.0 Setup

3.0.1 3.0.1 Imports & Dependencies

[1]:
```
//dependencies
%AddDeps org.vegas-viz vegas_2.11 0.3.11 --transitive
%AddDeps org.vegas-viz vegas-spark_2.11 0.3.11 --transitive

// https://mvnrepository.com/artifact/io.circe/circe-core
%AddDeps io.circe circe-core_2.11 0.11.2
%AddDeps io.circe circe-generic_2.11 0.11.2
%AddDeps io.circe circe-parser_2.11 0.11.2
```

```
Marking org.vegas-viz:vegas_2.11:0.3.11 for download
Obtained 42 files
Marking org.vegas-viz:vegas-spark_2.11:0.3.11 for download
Obtained 44 files
Marking io.circe:circe-core_2.11:0.11.2 for download
Obtained 2 files
Marking io.circe:circe-generic_2.11:0.11.2 for download
Obtained 2 files
Marking io.circe:circe-parser_2.11:0.11.2 for download
Obtained 2 files
```

[2]:
```
//imports
//spark
import org.apache.spark.ml.clustering.{LDA,LDAModel}
import org.apache.spark.ml.feature.{HashingTF, IDF, Tokenizer, CountVectorizer,␣
 ↪CountVectorizerModel}
```

```
import org.apache.spark.sql.{Row,SparkSession,DataFrame}
import org.apache.spark.ml.linalg.{DenseVector,SparseVector}
import org.apache.spark.sql.functions.udf

//Vegas
import vegas.sparkExt._
import vegas._
import vegas.data.External._
import vegas.DSL.SpecBuilder
implicit val render = vegas.render.ShowHTML(kernel.display.content("text/html",␣
↪_))

//etc
import java.io.File
import scala.reflect.io.Path
import java.io.PrintWriter
import scala.io.Source
import scala.concurrent.duration.Duration
import sys.process._
```

3.0.2 3.0.2 Initialize Spark

In the Apache Toree Kernel Spark parameters are passed via environmental variables or when creating the Kernel for the first time.

[3]:
```
val sparkSession = spark
import sparkSession.implicits._
```

3.0.3 3.0.3 Load the files and convert them into a dataframe

The english files are loaded using Spark's RDD `wholeTextFiles` method and later converted into a DataFrame using the `toDF` method. In this DataFrame the raw text is stored alongside the title of the book, which is obtained via the file name.

[ ]:
```
def load(path:String):DataFrame = {
    //loads the files and returns them as (title, content) pairs
    val dataRDD = sparkSession.sparkContext.wholeTextFiles(path)
    //map the paths to only the file names
    .map{case(fileName,content)=>(Path(fileName).name.replace(".
↪txt",""),content)}

    //convert them to a dataframe, naming the title column to 'label' and the␣
↪content to 'text'
    val dataDF = dataRDD.toDF("label","text")
    dataDF
}
//make it usable in compositions (andThen chaining)
```

```
val loadF = load _

val dataDF = load("data/en")
//display some
dataDF.show(5)
```

3.0.4 3.1 Text Segmentation

Only whitespace tokenization will be performed using Spark's machine learning (ML) `ml.feature.Tokenizer`.

[5]:
```
def tokenize(inputDF:DataFrame):DataFrame = {
    val tokenizer = new Tokenizer()
        .setInputCol("text")
        .setOutputCol("words")
    //tokenize using a standard whitespace tokenizer
    val tokenizedDF = tokenizer.transform(inputDF)
    tokenizedDF
}
val tokenizedDF = tokenize(dataDF)
tokenizedDF.show(5)
```

```
+--------------------+--------------------+--------------------+
|               label|                text|               words|
+--------------------+--------------------+--------------------+
|Captains Courageo...|Rudyard Kipling
...|[rudyard, kipling...|
|Dracula - Bram St...|Bram Stoker


...|[bram, stoker, , ...|
...|[jerome, k., jero...|
|Bush Boys, The - ...|Captain Mayne Rei...|[captain, mayne, ...|
|Macbeth - William...|William Shakespea...|[william, shakesp...|
+--------------------+--------------------+--------------------+
only showing top 5 rows
```

[5]:
```
[label: string, text: string ... 1 more field]
```

3.0.5 3.2 Preprocessing

All non-alphanumeric characters from the already lower case words will be removed. This is achieved using a user-defined function (*UDF*) and a simple regular expression.

[6]:
```
def preprocess(inputDF:DataFrame):DataFrame = {
    val removeNonAlphanumeric: Seq[String] => Seq[String] = _.map(
            word => word
            .replaceAll("[^A-z0-9]", ""))
        .filter(_.nonEmpty)
    val removeNonAlphanumericUDF = udf(removeNonAlphanumeric)
    val preprocessedDF = inputDF.
 ↪withColumn("words",removeNonAlphanumericUDF('words))
    preprocessedDF
}
val preprocessedDF = preprocess(tokenizedDF)
preprocessedDF.show(5)
//spelling mistake removal (very infrequent words) will be performed by the␣
 ↪CountVectorizer in the next step
```

```
+--------------------+--------------------+--------------------+
|               label|                text|               words|
+--------------------+--------------------+--------------------+
|Captains Courageo...|Rudyard Kipling
...|[rudyard, kipling...|
|Dracula - Bram St...|Bram Stoker


...|[bram, stoker, dr...|
...|[jerome, k, jerom...|
|Bush Boys, The - ...|Captain Mayne Rei...|[captain, mayne, ...|
|Macbeth - William...|William Shakespea...|[william, shakesp...|
+--------------------+--------------------+--------------------+
only showing top 5 rows
```

```
[6]: [label: string, text: string ... 1 more field]
```

3.0.6 3.3 Term Frequency – inverse Document Frequency (TF-IDF)

In this step the TF-IDF Matrix will be computed. This is a 2 step process: - First word counts will be computed using Spark ML's `ml.feature.CountVectorizer`. A minimum term frequency of 3 is used, in order to filter out potential misspellings. - Then then TF-IDF Matrix will be computed using Spark ML's `ml.feature.IDF`

```
[7]: case class CountsConfig(minTF: Int)
```

```
defined class CountsConfig
```

```
[ ]: //compute counts of words
def counts(inputDF: DataFrame, config: CountsConfig): (DataFrame, Array[String])␣
 ↪= {
    // get a CountVectorizerModel
    val cvModel: CountVectorizerModel = new CountVectorizer()
        .setInputCol("words")
        .setOutputCol("rawFeatures")
        //used atleast minTF times, to prevent spelling mistakes
        .setMinTF(config.minTF)
        .fit(inputDF)

    val vocabulary = cvModel.vocabulary
    val countsDF = cvModel.transform(inputDF)
    (countsDF, vocabulary)
}
val (countsDF,vocabulary) = counts(preprocessedDF,CountsConfig(3))
countsDF.show(5)
```

[9]:
```
def tfIdf(inputDF:DataFrame,vocabulary: Array[String]):(DataFrame,Array[String])␣
 ↪= {
        //use the already computed counts to build a TF-IDF model
    val tfidfModel = new IDF()
        .setInputCol("rawFeatures")
        .setOutputCol("features")
        .fit(inputDF)
    val TFIDFDF = tfidfModel.transform(inputDF)
    (TFIDFDF,vocabulary)
}
val (tfidfDF,_) = tfIdf(countsDF,vocabulary)
tfidfDF.select("label", "features").show(5)
```

```
+--------------------+--------------------+
|               label|            features|
+--------------------+--------------------+
|Captains Courageo...|(74467,[0,1,2,3,4...|
|Dracula - Bram St...|(74467,[0,1,2,3,4...|
|Three Men in a Bo...|(74467,[0,1,2,3,4...|
|Bush Boys, The - ...|(74467,[0,1,2,3,4...|
|Macbeth - William...|(74467,[0,1,2,3,4...|
+--------------------+--------------------+
only showing top 5 rows
```

[9]:
```
[label: string, text: string ... 3 more fields]
```

3.0.7 3.4 Latent Dirichlet allocation (LDA)

Perform LDA using Spark ML's `ml.feature.LDA`. $K = 10$ has been selected after experimenting with different Ks (3,5,10). If the algorithm doesn't converge befor that, at most 25 iterations will be performed.

[10]:
```
case class LDAConfig(k:Int, maxIter:Int)
```

```
defined class LDAConfig
```

[ ]:
```
def fitLDAModel(inputDF:DataFrame,vocabulary: Array[String], config:LDAConfig):
 ↪(LDAModel,Array[String]) = {
    val cachedInput = inputDF.cache
    // Trains a LDA model.
    val ldaModel = new LDA()
        .setK(config.k)
        .setMaxIter(config.maxIter)
        .fit(cachedInput)
    (ldaModel,vocabulary)
}

//train
val (ldaModel,_) = fitLDAModel(tfidfDF,vocabulary,LDAConfig(10,25))

// compute
val ldaTopicsDF = ldaModel.transform(tfidfDF)
```

3.0.8 3.5 Display Topics

In this section the produced topics will be explored and visualized using Vegas-viz.

[12]:
```
case class LDAEntry(label:String, dist: List[Double])
```

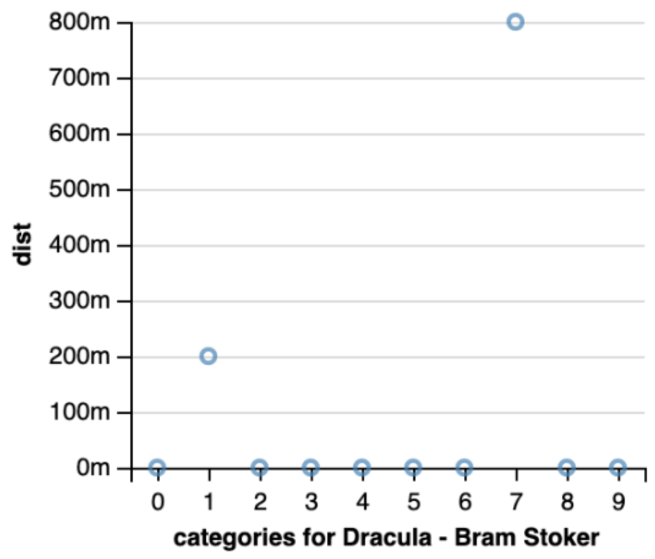
```
defined class LDAEntry
```

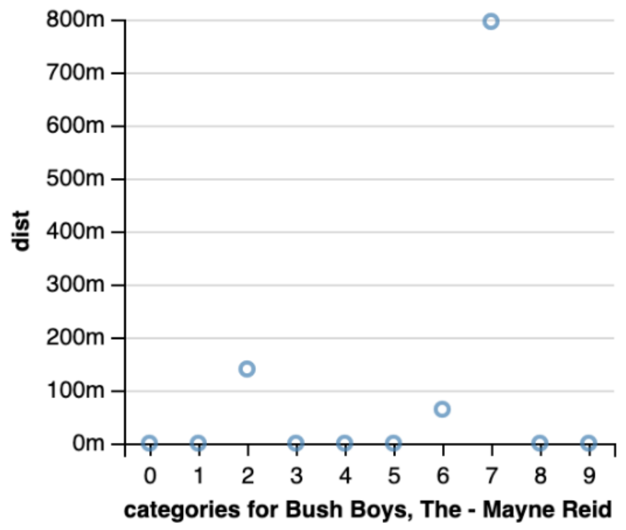
[ ]:
```
//convert to a case class for easier handling
val ldaDocumentsTopics = ldaTopicsDF
    .select("label","topicDistribution")
    .collect()
    .map(row => LDAEntry(row.getString(0), row.getAs[DenseVector](1).toArray.
 ↪toList))
```

3.5.1 Distribution for first 5 texts separately followed by all superimposed

[15]:
```
plot(dataToPlot.take(1))
```

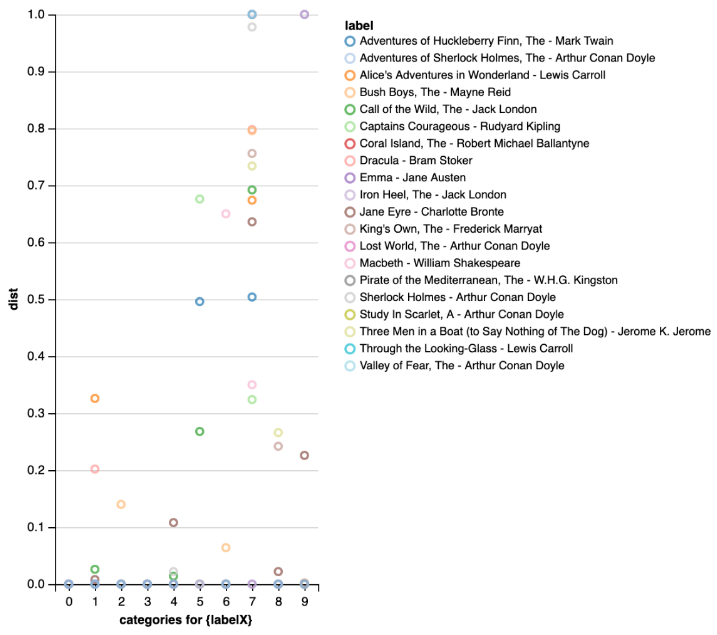
[16]:
```
plot(dataToPlot.drop(1).take(2))
val k = 10
plot2(d2.take(20 * k))
```


800m
700m
600m
500m
400m
300m
200m
100m
0m
dist
0 1 2 3 4 5 6 7 8 9
categories for Dracula - Bram Stoker


800m
700m
600m
500m
400m
300m
200m
100m
0m
dist
0 1 2 3 4 5 6 7 8 9
categories for Bush Boys, The - Mayne Reid

"

"



3.5.2 Top 5 terms of each topic

```
[ ]: //Describe topics by top N terms
     val topics = ldaModel.describeTopics(5)
     topics.show(false)
```

```
[20]: topics.map{
          case Row(id:Int,termIndices,termWeights) =>
              (id,termIndices.asInstanceOf[Seq[Int]].map(vocabulary(_)),termWeights.
        ↪asInstanceOf[Seq[Double]])
      }.show(false)
```

```
|0  |[thir, heavn, th, farr, hee]                  |[0.004627924442897164,
0.0024599481209023116, 0.0023531456189117136, 8.985575037830114E-4,
7.339605297358634E-4]  |
|1  |[larsen, oliver, bumble, sikes, fagin]        |[0.008593472903767307,
0.007941478340168623, 0.004790708209700835, 0.004700338034084353,
0.004201558864370088]    |
|2  |[calhoun, zeb, seguin, maurice, poindexter]   |[0.007044581529799887,
0.0068877247637948264, 0.00534622341310346, 0.00531466950174619,
0.005223113788151869]    |
|3  |[edna, cheditafa, judith, shirley, deerslayer]  |[0.004956025753691830 5,
0.0022328761913813252, 0.002079438764203889, 0.0018911030265342807,
0.0016853971569470796]|
|4  |[hepzibah, phoebe, pyncheon, clifford, darcy]  |[0.005664168070672671,
0.004589651998766695, 0.00385792137784414, 0.0033651293235921957,
0.001742477681908681]    |
|5  |[harvey, dan, warnt, disko, salters]          |[0.005201115306385349,
0.0036358710158439236, 0.0032916483883781954, 0.0025873838701126017,
0.0018361154550118994]|
|6  |[oliver, macbeth, ada, sikes, fagin]              |[0.004471984896169522,
0.00426690698768879, 0.002161016272866552, 0.0020882713125453672,
0.0019372436495667265]   |
|7  |[holmes, alice, watson, judith, deerslayer]       |[0.006722142638052459,
0.0028120335239338688, 0.002026575708152155, 0.001567725611565608,
0.0015171274859362709]  |
|8  |[harris, mackellar, mcelvina, rainscourt, seymour]|[0.0037748861573694893,
0.0037112673568695555, 0.002657441326408293, 0.0025329446878162177,
0.0022395318830612104]|
|9  |[emma, harriet, weston, deerslayer, knightley]    |[0.01782220132679931,
0.009323682976361342, 0.008648645301828815, 0.008540896113920603,
0.007995067976542912]     |
```

3.5.3 Similarity How to read the graph: when hovering over a node, it and it's connections will be highligted. The length of an edge denotes the similarity between 2 nodes (the shorter, more similar) and is in proportion to all other edges of the node.

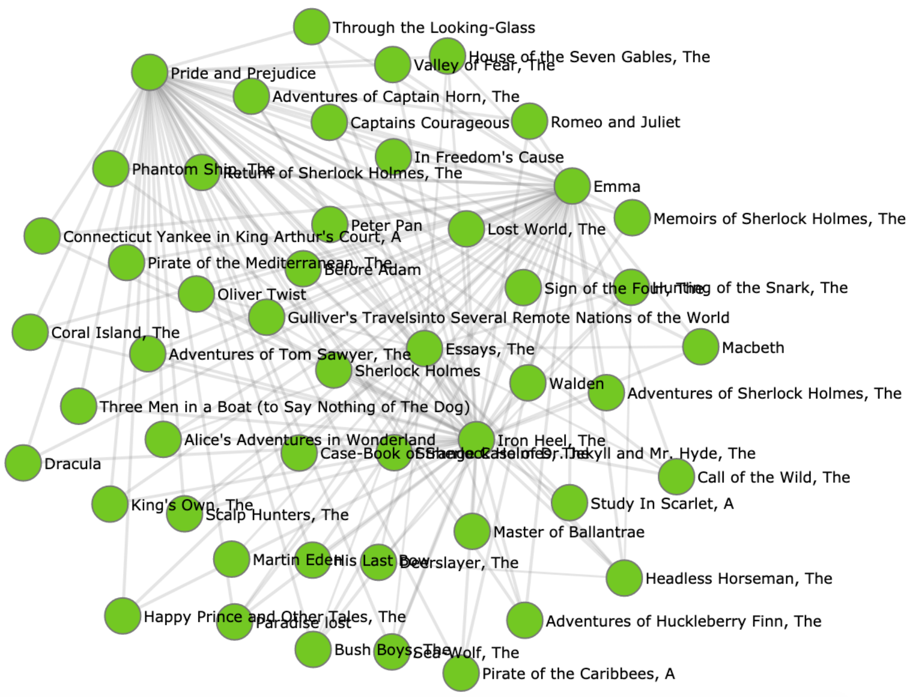
[21]:
```
//define needed functions
var log2 = (x: Double) => math.log10(x) / math.log10(2.0)

//https://en.wikipedia.org/wiki/Jensen%E2%80%93Shannon_divergence
def jensenShannonDivergence(x: List[Double], y: List[Double]): Double = {
  val xPlusy = x.zip(y).map { case (a, b) => a + b }
  val d1 = x.zip(xPlusy).map { case (x, xPlusY) => x * log2((2.0 * x) / xPlusY) }
  val d2 = y.zip(xPlusy).map { case (y, xPlusY) => y * log2((2.0 * y) / xPlusY) }
  val d = 0.5 * (d1.sum + d2.sum)
  d
}
```

[21]:
```
> Double = <function1>
```

[ ]:
```
//convert to required json format
val jsonData = {
        val combinations = ldaDocumentsTopics.combinations(2).
 ↪map(x=>(x(0),(x(1))))
    val res = combinations
        .map{case(a,b)=>(a.label,b.label,jensenShannonDivergence(a.dist,b.dist))}
        .filter{case(a,b,distance)=> distance > 0.7}
        .toList

    case class Node(name:String,display:String, id:Int)

    case class Link(source:Int, target:Int, strength:Double)

    case class Data(nodes:List[Node],links:List[Link])

    val nodes = res
        .flatMap{case(a,b,_)=>List(a,b)}
        .distinct
        .zipWithIndex
        .map{case(x,i)=>Node(x,x.substring(0,x.lastIndexOf("-")),i)}
    val lookup = nodes.map(x=>x.name->x.id).toMap
    val links = res.
 ↪map{case(a,b,distance)=>Link(lookup(a),lookup(b),f"$distance%2.2f".toDouble)}
    val data = Data(nodes,links)

    import io.circe._, io.circe.generic.auto._, io.circe.parser._, io.circe.
 ↪syntax._
    data.asJson.noSpaces
}
```

```
[23]: //fill the template and write to a new file, which is then displayed in an␣
       ↪iframe.
      //Simply writing as json and loading the data there doesn't work sadly, since␣
       ↪standalone html pages can't reference
      // the local file system
      val iframe = {
          val writer = new PrintWriter(new File("index.html"))
          val newContent = Source.fromFile("Template.html").getLines.map(x=>x.
       ↪replace("@HERE",jsonData))
          newContent.foreach(x=>writer.write(x+"\n"))
          writer.close()
          val iframe ="""
          <iframe width ="1050" height="850" src="index.html"></iframe>
          """
          iframe
      }
```

```
[24]: kernel.display.content("text/html", iframe)
```


Through the Looking-Glass
House of the Seven Gables, The
Valley of Fear, The
Pride and Prejudice
Adventures of Captain Horn, The
Captains Courageous
Romeo and Juliet
In Freedom's Cause
Phantom Ship, The
Return of Sherlock Holmes, The
Emma
Memoirs of Sherlock Holmes, The
Peter Pan
Lost World, The
Connecticut Yankee in King Arthur's Court, A
Pirate of the Mediterranean, The
Before Adam
Sign of the Four, The
Hunting of the Snark, The
Oliver Twist
Gulliver's Travelsinto Several Remote Nations of the World
Coral Island, The
Essays, The
Macbeth
Adventures of Tom Sawyer, The
Sherlock Holmes
Walden
Adventures of Sherlock Holmes, The
Three Men in a Boat (to Say Nothing of The Dog)
Alice's Adventures in Wonderland
Iron Heel, The
Case-Book of Sherlock Holmes, The
Strange Case of Dr. Jekyll and Mr. Hyde, The
Dracula
Call of the Wild, The
King's Own, The
Scalp Hunters, The
Study In Scarlet, A
Master of Ballantrae
Martin Eden
His Last Bow
Deerslayer, The
Headless Horseman, The
Happy Prince and Other Tales, The
Paradise lost
Adventures of Huckleberry Finn, The
Bush Boys, The
Sea-Wolf, The
Pirate of the Caribbees, A

4 4. Tests

In the project using scalatest and scalactic. run `cd tfidflda && sbt test` in order to start them. Sadly this is not yet supported by the Apache Toree kernel.

5 5. Runtime analysis

[ ]:
```
val sparkOptions = List(
        "--master local[*] --executor-memory 256MB --driver-memory 1G␣
↪--num-executors 1 --executor-cores 4",
        "--master local[*] --executor-memory 1G --driver-memory 1G␣
↪--num-executors 1 --executor-cores 16",
        "--master local[*] --executor-memory 1G --driver-memory 1G␣
↪--num-executors 1 --executor-cores 4",
        "--master local[*] --executor-memory 1G --driver-memory 1G␣
↪--num-executors 2 --executor-cores 8",
        "--master local[*] --executor-memory 512MB --driver-memory 1G␣
↪--num-executors 4 --executor-cores 4"
    )

val results = {
    val dataPath = Path("data/en").toAbsolute
    sparkOptions.map(
        sparkOption => time(assert((f"spark-submit ${sparkOption} --class de.htw.
↪ai.progkonz.SparkApp main.jar ${dataPath}" !) == 0))
    ).toList
}
```

[27]:
```
sparkOptions
    .zip(results)
    .foreach{case(sparkConf,(elapsedNanoseconds,_))=>
        println(f"[${Duration.fromNanos(elapsedNanoseconds).toSeconds} s.]␣
↪$sparkConf")}
```

```
[103 s.] --master local[*] --executor-memory 256MB --driver-memory 1G --num-
executors 1 --executor-cores 4
[109 s.] --master local[*] --executor-memory 1G --driver-memory 1G --num-
executors 1 --executor-cores 16
[96 s.] --master local[*] --executor-memory 1G --driver-memory 1G --num-
executors 1 --executor-cores 4
[96 s.] --master local[*] --executor-memory 1G --driver-memory 1G --num-
executors 2 --executor-cores 8
[101 s.] --master local[*] --executor-memory 512MB --driver-memory 1G --num-
executors 4 --executor-cores 4
```