The data set comes from [55000+ Song Lyrics](https://www.kaggle.com/mousehead/songlyrics), which contains over 55,000+ songs. It is a data frame with 55,000+ rows and four columns:

* **artist**
* **song**
* **link**
* **text**

library(igraph)

library(ggraph)

library(ggplot2)

library(wordcloud2)

library(dplyr)

library(widyr)

library(tidytext)

library(tm)

library(stringr)

library(topicmodels)

library(reshape2)

library(quanteda)

library(Rtsne)

library(DT)

library(knitr)

library(animation)

library(ldatuning)

set.seed(201712)

# Preperation -------------------------------------------------------------

songs <- read.csv("songdata.csv")

songs$song <- songs$song %>% as.character()

songs$artist <- songs$artist %>% as.character()

songs$link <- songs$link %>% as.character()

songs$text <- songs$text %>% as.character()

Our goal is to perform a comprehensive analysis of the song texts to identify the Christmas songs. In order to do so, first we add an additional column to the data frame to give each song a **label** of either **Christmas** or **Not Christmas**, where every song which contains the words **Christmas**, **Xmas** or **X-mas** will be labeled as **Christmas** and otherwise as **Not Christmas**.

# Initialization of the Labels

label <- character(dim(songs)[1])

for(i in 1:dim(songs)[1]){

if(str\_detect(songs$song[i], "Christmas") |

str\_detect(songs$song[i], "X-mas") |

str\_detect(songs$song[i], "Xmas")){

label[i] <- "Christmas"

} else{

label[i] <- "Not Christmas"

}

}

songs <- songs %>%

mutate(Label = label)

This is just the initialization of the labels, later we will apply Naive Bayes to a training set to identify the other Christmas songs. First of all, we will start by exploring the data set by means of some intuitive descriptive approaches.

D3Vis <- function(edgeList, directed){

colnames(edgeList) <- c("SourceName", "TargetName", "Weight")

# Min-Max & Inverse scaling, because the weights should represent distance/similarity

edgeList$Weight <- 1 - edgeList$Weight

weight.min <- edgeList$Weight %>% min

weight.max <- edgeList$Weight %>% max

edgeList$Weight <- (edgeList$Weight - weight.min)/(weight.max - weight.min)

# Create a graph. Use simplyfy to ensure that there are no duplicated edges or self loops

gD <- igraph::simplify(igraph::graph.data.frame(edgeList, directed=directed))

# Create a node list object (actually a data frame object) that will contain information about nodes

nodeList <- data.frame(ID = c(0:(igraph::vcount(gD) - 1)), # because networkD3 library requires IDs to start at 0

nName = igraph::V(gD)$name)

# Map node names from the edge list to node IDs

getNodeID <- function(x){

which(x == igraph::V(gD)$name) - 1 # to ensure that IDs start at 0

}

# And add them to the edge list

edgeList <- plyr::ddply(edgeList, .variables = c("SourceName", "TargetName", "Weight"),

function (x) data.frame(SourceID = getNodeID(x$SourceName),

TargetID = getNodeID(x$TargetName)))

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++#

# Calculate some node properties and node similarities that will be used to illustrate

# different plotting abilities and add them to the edge and node lists

# Calculate degree for all nodes

nodeList <- cbind(nodeList, nodeDegree=igraph::degree(gD, v = igraph::V(gD), mode = "all"))

# Calculate betweenness for all nodes

betAll <- igraph::betweenness(gD, v = igraph::V(gD), directed = directed) / (((igraph::vcount(gD) - 1) \* (igraph::vcount(gD)-2)) / 2)

betAll.norm <- (betAll - min(betAll))/(max(betAll) - min(betAll))

nodeList <- cbind(nodeList, nodeBetweenness=100\*betAll.norm) # We are scaling the value by multiplying it by 100 for visualization purposes only (to create larger nodes)

rm(betAll, betAll.norm)

#Calculate Dice similarities between all pairs of nodes

dsAll <- igraph::similarity.dice(gD, vids = igraph::V(gD), mode = "all")

F1 <- function(x) {data.frame(diceSim = dsAll[x$SourceID +1, x$TargetID + 1])}

edgeList <- plyr::ddply(edgeList, .variables=c("SourceName", "TargetName", "Weight", "SourceID", "TargetID"),

function(x) data.frame(F1(x)))

rm(dsAll, F1, getNodeID, gD)

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++#

# We will also create a set of colors for each edge, based on their dice similarity values

# We'll interpolate edge colors based on the using the "colorRampPalette" function, that

# returns a function corresponding to a collor palete of "bias" number of elements (in our case, that

# will be a total number of edges, i.e., number of rows in the edgeList data frame)

F2 <- colorRampPalette(c("#FFFF00", "#FF0000"), bias = nrow(edgeList), space = "rgb", interpolate = "linear")

colCodes <- F2(length(unique(edgeList$diceSim)))

edges\_col <- sapply(edgeList$diceSim, function(x) colCodes[which(sort(unique(edgeList$diceSim)) == x)])

rm(colCodes, F2)

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++#

# revise transformation of the weights

edgeList$Weight <- -(edgeList$Weight\*(weight.max - weight.min) + weight.min - 1)

# Let's create a network

D3\_network\_LM <- networkD3::forceNetwork(Links = edgeList, # data frame that contains info about edges

Nodes = nodeList, # data frame that contains info about nodes

Source = "SourceID", # ID of source node

Target = "TargetID", # ID of target node

Value = "Weight", # value from the edge list (data frame) that will be used to value/weight relationship amongst nodes

NodeID = "nName", # value from the node list (data frame) that contains node description we want to use (e.g., node name)

Nodesize = "nodeBetweenness", # value from the node list (data frame) that contains value we want to use for a node size

Group = "nodeDegree", # value from the node list (data frame) that contains value we want to use for node color

# height = 500, # Size of the plot (vertical)

# width = 1000, # Size of the plot (horizontal)

fontSize = 20, # Font size

linkDistance = networkD3::JS("function(d) { return 10\*d.value; }"), # Function to determine distance between any two nodes, uses variables already defined in forceNetwork function (not variables from a data frame)

# linkWidth = networkD3::JS("function(d) { return d.value/5; }"),# Function to determine link/edge thickness, uses variables already defined in forceNetwork function (not variables from a data frame)

opacity = 0.85, # opacity

arrows = directed,

zoom = TRUE, # ability to zoom when click on the node

# opacityNoHover = 0.1, # opacity of labels when static

legend = F,

linkColour = edges\_col) # edge colors

# Plot network

D3\_network\_LM

}

**Exploration of the initial Christmas songs**

***Cleaning & Tokenization***

We should start with the data cleaning and tokenization. Afterwards, the Christmas songs will be selected and saved as a variable.

songs.unnest <- songs %>%

unnest\_tokens(word, text) %>%

anti\_join(tibble(word = stop\_words$word)) %>%

filter(!str\_detect(word, "\\d+"))

xmas.unnest <- songs.unnest %>%

filter(Label == "Christmas")

***Correlation Analysis***

Now we can start analyzing the initial Christmas songs by means of correlations from different perspectives. In the following, we will visualize the correlations with the networkD3 html widget where nodes with the same total number of connections will be given the same color and the color of the edge implies the number of common neighbors shared by two nodes. Moreover, the size of a node indicates the centrality of it, which is defined by the betweenness, i.e. the number of shortest paths going through it. Where the distance between two nodes is the minimum maximum transformation of 1 minus the correlation, which makes sense because intuitively the higher the correlation, the nearer two nodes should be. Moreover, the shorter the distance, the wider the edge.

Note that the correlations are always based on lyrics.

***Correlation between words***

The correlation between words which appeared more than 100 times and are correlated with at least one other word with a correlation greater than 0.55.

correlation.words <- xmas.unnest %>%

group\_by(word) %>%

filter(n() > 100) %>%

ungroup() %>%

pairwise\_cor(word, song, sort = T)

# Network visualization

correlation.words %>%

filter(correlation > 0.55) %>%

D3Vis(directed = F)

***Correlation between songs***

The correlation between songs which are correlated with at least 3 other songs with a correlation greater than 0.75. With this, we may detect similiar or just slightly modified songs.

correlation.songs <- xmas.unnest %>%

pairwise\_cor(song, word, sort = T)

# Network visualization

correlation.songs %>%

filter(correlation > 0.75) %>%

group\_by(item1) %>%

filter(n() >= 3) %>%

ungroup() %>%

D3Vis(directed = F)

***Correlation between certain words***

The correlation between certain words

correlation.words %>%

filter(item1 == "christus" |

item1 == "jesus" |

item1 == "snow" |

item1 == "reindeer" |

item1 == "home" |

item1 == "holy" |

item1 == "love" |

item1 == "tree" |

item1 == "white" |

item1 == "christmas",

correlation > 0.4) %>%

D3Vis(directed = F)

***Correlation between artists***

The correlation between artists

correlation.artists <- xmas.unnest %>%

pairwise\_cor(artist, word, sort = T)

# Network Visualization

correlation.artists %>%

filter(correlation > 0.8) %>%

group\_by(item1) %>%

filter(n() >= 3) %>%

ungroup() %>%

D3Vis(directed = F)

***Word Cloud***

Word cloud of the initial Christmas songs

xmas.cloud <- xmas.unnest %>%

count(word) %>%

as.data.frame()

xmas.cloud %>%

wordcloud2(minSize = 3, shape = 'star')

**Naive Bayes**

Naive Bayes is a popular supervised machine learning algorithm to handle classification problems with a huge amount of features. It is „naive“ in the sense that, conditioned on a class, the features are assumed to be independently distributed. In our case, we would like to know, given a bunch of features, i.e. the [tf-idf](https://en.wikipedia.org/wiki/Tf%E2%80%93idf" \t "_blank) of words in a document, whether a song should be classified as Christmas song or not by Naive Bayes.

Generally, given features $\mathbf{x} = (x\_1, …, x\_p)$ we have $$\begin{aligned} \mathbb{P}(C\_k|\mathbf{x}) &= \frac{\mathbb{P}(C\_k)\mathbb{P}(\mathbf{x}|C\_k)}{\mathbb{P(\mathbf{x})}} \ &= \frac{\mathbb{P}(C\_k)\prod\_{i = 1}^p\mathbb{P}(x\_i|C\_k)}{\mathbb{P(\mathbf{x})}} \varpropto \mathbb{P}(C\_k)\prod\_{i = 1}^p\mathbb{P}(x\_i|C\_k) \end{aligned} $$ Where $\mathbb{P}(C\_k)$ is called the *prior* and $\mathbb{P}(C\_k|\mathbf{x})$ the *posterior* and $\mathbb{P}(\mathbf{x}|C\_k)$ the *likelihood*. The MLE is obviously $$\hat{C}:= \underset{k}{\arg \max},\mathbb{P}(C\_k)\prod\_{i = 1}^p\mathbb{P}(x\_i|C\_k)$$

Because we assume that the features are independent conditioning on an arbitrary class. We may therefore estimate $\mathbb{P}(x\_i|C\_k), \forall i = 1,…, p$ independently of other features using a training set, which makes the whole thing much easier. The popular assumptions of the likelihood are Gaussian, multinomial or Bernoulli. The harder part of constructing the maximum likelihood estimator is the choice of the prior distribution, i.e. the probability distribution of the classes. Where it is usually assumed to be uniformly distributed or estimated by the class frequencies. In our case the multinomial distribution for the likelihood and the uniform distribution for the prior are used, which means we have no prejudice regarding the categorization of the songs without given further information.

***Identify the hidden Christmas songs***

# Document Feature Matrix

songs.dfm.tfidf <- corpus(songs, text\_field = "text",

docid\_field = "song") %>%

dfm(tolower = T,

stem = TRUE,

remove\_punct = TRUE,

remove = stopwords("english")) %>%

dfm\_trim(min\_count = 5, min\_docfreq = 3) %>%

dfm\_weight(type = "tfidf")

# Determine the Indizes for the training set

christmas.index <- which(label == "Christmas")

not\_christmas.index <- which(label == "Not Christmas")

christmas.train.index <- christmas.index

not\_christmas.train.index <- sample(not\_christmas.index, length(christmas.index))

train.index <- c(christmas.train.index, not\_christmas.train.index)

label.train <- label[train.index]

trainning.set <- songs.dfm.tfidf[train.index, ]

# Train the Model

classifier\_NB <- textmodel\_NB(trainning.set, label.train)

# Prediction

predictions <- classifier\_NB %>%

predict(newdata = songs.dfm.tfidf)

# Confusion Matrix

confusion <- table(predictions$nb.predicted, label)

confusion

So we have identified 2965 hidden Christmas songs and there are 2 songs out of the initial 500 Christmas songs that are rejected by Naive Bayes as Christmas songs.

***Explore the hidden Christmas songs***

#Determine the Indizes for the hidden (not) Christmas Songs.

hidden.index <- (predictions$nb.predicted == "Christmas") & (songs$Label == "Not Christmas")

hidden\_not.index <- (predictions$nb.predicted == "Not Christmas") & (songs$Label == "Christmas")

# Change the labels

label[hidden.index] <- "Hidden Christmas"

label[hidden\_not.index] <- "Hidden Not Christmas"

songs$Label <- label

songs.dfm.tfidf@docvars$Label <- label

# Wordcloud for the hidden Christmas Songs

hidden.xmas <- songs[hidden.index, ]

hidden.unnest <- hidden.xmas %>%

unnest\_tokens(word, text) %>%

anti\_join(tibble(word = stop\_words$word)) %>%

filter(!str\_detect(word, "\\d+"))

hidden.unnest %>%

count(word) %>%

filter(n >= 5) %>%

as.data.frame() %>%

wordcloud2(shape = "star", minSize = 5)

# Correlation

hidden.correlation.words <- hidden.unnest %>%

group\_by(word) %>%

filter(n() > 15) %>%

ungroup() %>%

pairwise\_cor(word, song, sort = T)

# Network visualization

hidden.correlation.words %>%

filter(correlation > 0.65) %>%

group\_by(item1) %>%

filter(n() >= 20) %>%

ungroup() %>%

D3Vis(directed = F)

We have therefore successfully identified a bunch of religous christmas songs, whose titles usually do not contain the word „Christmas“ or „X-mas“.

**Latent Dirichtlet Allocation & t-Statistics Stochastic Neighbor Embedding**

***Data Preparation***

Only the top 300 features for the Christmas songs including the hidden ones will be used to calculate the Rtsne & LDA, else the memory space will not be sufficient.

xmas.dfm.tfidf <- songs.dfm.tfidf %>%

dfm\_subset(Label == "Christmas" | Label == "Hidden Christmas")

songs.dfm.tfidf\_300 <- songs.dfm.tfidf %>%

dfm\_select(pattern = xmas.dfm.tfidf %>%

topfeatures(300) %>%

names(), selection = "keep")

xmas.dfm.tfidf\_300 <- xmas.dfm.tfidf %>%

dfm\_select(pattern = xmas.dfm.tfidf %>%

topfeatures(300) %>%

names(), selection = "keep")

NetworkD3 Package Details

For very basic [force directed network](http://en.wikipedia.org/wiki/Force-directed_graph_drawing) graphics you can use simpleNetwork. For example:

*# Load package*

**library**(networkD3)

*# Create fake data*

src <- **c**("A", "A", "A", "A",

"B", "B", "C", "C", "D")

target <- **c**("B", "C", "D", "J",

"E", "F", "G", "H", "I")

networkData <- **data.frame**(src, target)

*# Plot*

**simpleNetwork**(networkData)

ABCDEFGHIJ

**[> forceNetwork](http://christophergandrud.github.io/networkD3/" \l "force)**

Use forceNetwork to have more control over the appearance of the forced directed network and to plot more complicated networks. Here is an example:

*# Load data*

**data**(MisLinks)

**data**(MisNodes)

*# Plot*

**forceNetwork**(Links = MisLinks, Nodes = MisNodes,

Source = "source", Target = "target",

Value = "value", NodeID = "name",

Group = "group", opacity = 0.8)

MyrielNapoleonMlle.BaptistineMme.MagloireCountessdeLoGeborandChamptercierCravatteCountOldManLabarreValjeanMargueriteMme.deRIsabeauGervaisTholomyesListolierFameuilBlachevilleFavouriteDahliaZephineFantineMme.ThenardierThenardierCosetteJavertFaucheleventBamataboisPerpetueSimpliceScaufflaireWoman1JudgeChampmathieuBrevetChenildieuCochepaillePontmercyBoulatruelleEponineAnzelmaWoman2MotherInnocentGribierJondretteMme.BurgonGavrocheGillenormandMagnonMlle.GillenormandMme.PontmercyMlle.VauboisLt.GillenormandMariusBaronessTMabeufEnjolrasCombeferreProuvaireFeuillyCourfeyracBahorelBossuetJolyGrantaireMotherPlutarchGueulemerBabetClaquesousMontparnasseToussaintChild1Child2BrujonMme.Hucheloup

From version 0.1.3 you can also allow scroll-wheel zooming by setting zoom = TRUE.

**[> sankeyNetwork](http://christophergandrud.github.io/networkD3/" \l "sankey)**

You can also create Sankey diagrams with sankeyNetwork. Here is an example using downloaded JSON data:

*# Load energy projection data*

*# Load energy projection data*

URL <- **paste0**(

"<https://cdn.rawgit.com/christophergandrud/networkD3/>",

"master/JSONdata/energy.json")

Energy <- jsonlite::**fromJSON**(URL)

*# Plot*

**sankeyNetwork**(Links = Energy$links, Nodes = Energy$nodes, Source = "source",

Target = "target", Value = "value", NodeID = "name",

units = "TWh", fontSize = 12, nodeWidth = 30)

Agricultural 'waste'Bio-conversionLiquidLossesSolidGasBiofuel importsBiomass importsCoal importsCoalCoal reservesDistrict heatingIndustryHeating and cooling - commercialHeating and cooling - homesElectricity gridOver generation / exportsH2 conversionRoad transportAgricultureRail transportLighting & appliances - commercialLighting & appliances - homesGas importsNgasGas reservesThermal generationGeothermalH2HydroInternational shippingDomestic aviationInternational aviationNational navigationMarine algaeNuclearOil importsOilOil reservesOther wastePumped heatSolar PVSolar ThermalSolarTidalUK land based bioenergyWaveWind

**[> radialNetwork](http://christophergandrud.github.io/networkD3/" \l "radial)**

From version 0.2, tree diagrams can be created using radialNetwork or diagonalNetwork.

URL <- **paste0**(

"<https://cdn.rawgit.com/christophergandrud/networkD3/>",

"master/JSONdata//flare.json")

## Convert to list format

Flare <- jsonlite::**fromJSON**(URL, simplifyDataFrame = FALSE)

*# Use subset of data for more readable diagram*

Flare$children = Flare$children[1:3]

**radialNetwork**(List = Flare, fontSize = 10, opacity = 0.9)

flareanalyticsanimatedataclustergraphoptimizationEasingFunctionSequenceinterpolateISchedulableParallelPauseSchedulerSequenceTransitionTransitionerTransitionEventTweenconvertersDataFieldDataSchemaDataSetDataSourceDataTableDataUtilAgglomerativeClusterCommunityStructureHierarchicalClusterMergeEdgeBetweennessCentralityLinkDistanceMaxFlowMinCutShortestPathsSpanningTreeAspectRatioBankerArrayInterpolatorColorInterpolatorDateInterpolatorInterpolatorMatrixInterpolatorNumberInterpolatorObjectInterpolatorPointInterpolatorRectangleInterpolatorConvertersDelimitedTextConverterGraphMLConverterIDataConverterJSONConverter

**diagonalNetwork**(List = Flare, fontSize = 10, opacity = 0.9)

flareanalyticsanimatedataclustergraphoptimizationEasingFunctionSequenceinterpolateISchedulableParallelPauseSchedulerSequenceTransitionTransitionerTransitionEventTweenconvertersDataFieldDataSchemaDataSetDataSourceDataTableDataUtilAgglomerativeClusterCommunityStructureHierarchicalClusterMergeEdgeBetweennessCentralityLinkDistanceMaxFlowMinCutShortestPathsSpanningTreeAspectRatioBankerArrayInterpolatorColorInterpolatorDateInterpolatorInterpolatorMatrixInterpolatorNumberInterpolatorObjectInterpolatorPointInterpolatorRectangleInterpolatorConvertersDelimitedTextConverterGraphMLConverterIDataConverterJSONConverter

**[> dendroNetwork](http://christophergandrud.github.io/networkD3/" \l "dendro)**

From version 0.2, it is also possible to create dendrograms using dendroNetwork.

hc <- **hclust**(**dist**(USArrests), "ave")

**dendroNetwork**(hc, height = 600)

FloridaNorth CarolinaCaliforniaHawaiiMarylandAlaskaWashingtonRhode IslandMissouriGeorgiaIdahoArizonaNew MexicoDelawareMississippiSouth CarolinaOregonMassachusettsNew JerseyArkansasTennesseeColoradoTexasNebraskaOhioUtahWest VirginiaAlabamaLouisianaIllinoisNew YorkMichiganNevadaWyomingKentuckyMontanaIndianaKansasConnecticutPennsylvaniaMaineSouth DakotaNorth DakotaVermontMinnesotaOklahomaVirginiaWisconsinIowaNew Hampshire

**Interacting with igraph**

You can use igraph to create network graph data that can be plotted with **networkD3**. The igraph\_to\_networkD3 function converts igraph graphs to lists that work well with **networkD3**. For example:

*# Load igraph*

**library**(igraph)

*# Use igraph to make the graph and find membership*

karate <- **make\_graph**("Zachary")

wc <- **cluster\_walktrap**(karate)

members <- **membership**(wc)

*# Convert to object suitable for networkD3*

karate\_d3 <- **igraph\_to\_networkD3**(karate, group = members)

*# Create force directed network plot*

**forceNetwork**(Links = karate\_d3$links, Nodes = karate\_d3$nodes,

Source = 'source', Target = 'target',

NodeID = 'name', Group = 'group')

12345678910111213141516171819202122232425262728293031323334

**Output**

**Saving to an external stand alone HTML file**

Use saveNetwork to save a network to a stand alone HTML file:

**library**(magrittr)

**simpleNetwork**(networkData) %>%

**saveNetwork**(file = 'Net1.html')

**Including in an RMarkdown file**

It is simple to include a **networkD3** graphic in an RMarkdown file. Simply place the code to create the graph in a code chunk the same way you would any other plot.

**Including in Shiny web apps**

You can also easily include **networkD3** graphs in Shiny web apps.

In the *server.R* file create the graph by placing the function inside of render\*Network, where the \* is either Simple, Force, or Sankey depending on the graph type. For example:

output$force <- **renderForceNetwork**({

**forceNetwork**(Links = MisLinks, Nodes = MisNodes,

Source = "source", Target = "target",

Value = "value", NodeID = "name",

Group = "group", opacity = input$opacity)

})

===========================================================================

***LDA***

LDA stands for Latent Dirichtlet Allocation, which was introduced in Blei, Ng, Jordan (2003). It is a generative probabilistic model of a corpus, where the documents are represented as random mixtures over latent topics and for a single document there are usually only a few topics that are assigned unneglectable probabilities. Moreover, each topic is characterized by a distribution over words, where usually only a small set of words will be assigned significant probabilities for a certain topic. Either the variational expectation maximization algorithm or Gibbs sampling is used for the statistical inference of the parameters.

LDA requires a fixed number of topics, i.e. it assumes that the number of topics should already be known before applying the algorithm. However, there are possibilities to determine the optimal number of topics by different performance metrics.

install.packages("ldatuning")

or downloaded from the GitHub repository (developer version).

install.packages("devtools")

library("nikita-moor/ldatuning")

Package ldatuning realizes 4 metrics to select perfect number of topics for LDA model.

**library**("ldatuning")

Load “AssociatedPress” dataset from the topicmodels package.

**library**("topicmodels")

data("AssociatedPress", package="topicmodels")

dtm <- AssociatedPress[1:10, ]

The most easy way is to calculate all metrics at once. All existing methods require to train multiple LDA models to select one with the best performance. It is computation intensive procedure and ldatuning uses parallelism, so do not forget to point correct number of CPU cores in mc.core parameter to archive the best performance.

All standard LDA methods and parameters from topimodels package can be set with method and control.

result <- FindTopicsNumber(

dtm,

topics = seq(from = 2, to = 15, by = 1),

metrics = c("Griffiths2004", "CaoJuan2009", "Arun2010", "Deveaud2014"),

method = "Gibbs",

control = list(seed = 77),

mc.cores = 2L,

verbose = TRUE

)

## fit models... done.

## calculate metrics:

## Griffiths2004... done.

## CaoJuan2009... done.

## Arun2010... done.

## Deveaud2014... done.

Result is a number of topics and corresponding values of metrics

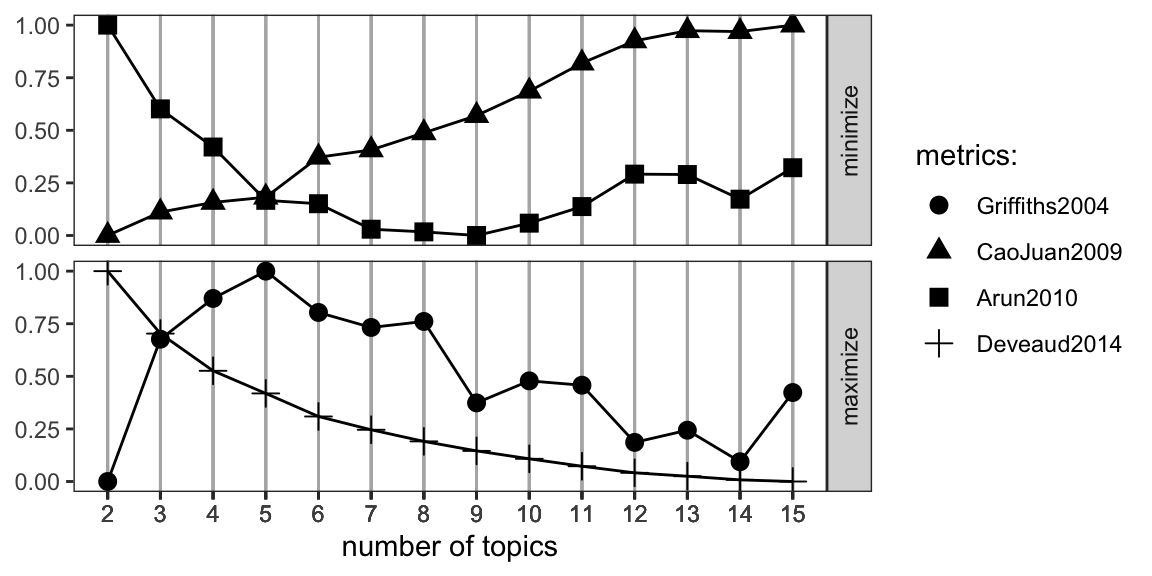
| **topics** | **Griffiths2004** | **CaoJuan2009** | **Arun2010** | **Deveaud2014** |
| --- | --- | --- | --- | --- |
| 15 | -15297.82 | 0.5047240 | 15.92711 | 0.1362596 |
| 14 | -15338.24 | 0.4927860 | 15.36552 | 0.1406462 |
| 13 | -15319.82 | 0.4944709 | 15.80569 | 0.1504368 |
| 12 | -15326.94 | 0.4756351 | 15.81278 | 0.1594651 |
| 11 | -15293.55 | 0.4347111 | 15.23313 | 0.1770861 |
| 10 | -15291.00 | 0.3829542 | 14.93706 | 0.1969989 |
| 9 | -15303.87 | 0.3379840 | 14.71664 | 0.2181424 |
| 8 | -15256.30 | 0.3061726 | 14.78140 | 0.2435689 |
| 7 | -15259.80 | 0.2746812 | 14.82908 | 0.2746203 |
| 6 | -15251.04 | 0.2612029 | 15.28425 | 0.3101625 |
| 5 | -15226.91 | 0.1875260 | 15.34470 | 0.3718687 |
| 4 | -15242.86 | 0.1779016 | 16.29708 | 0.4323482 |
| 3 | -15266.66 | 0.1600736 | 16.97832 | 0.5318997 |
| 2 | -15349.79 | 0.1169522 | 18.47430 | 0.6989189 |

Simple approach in analyze of metrics is to find extremum, more complete description is in corresponding papers:

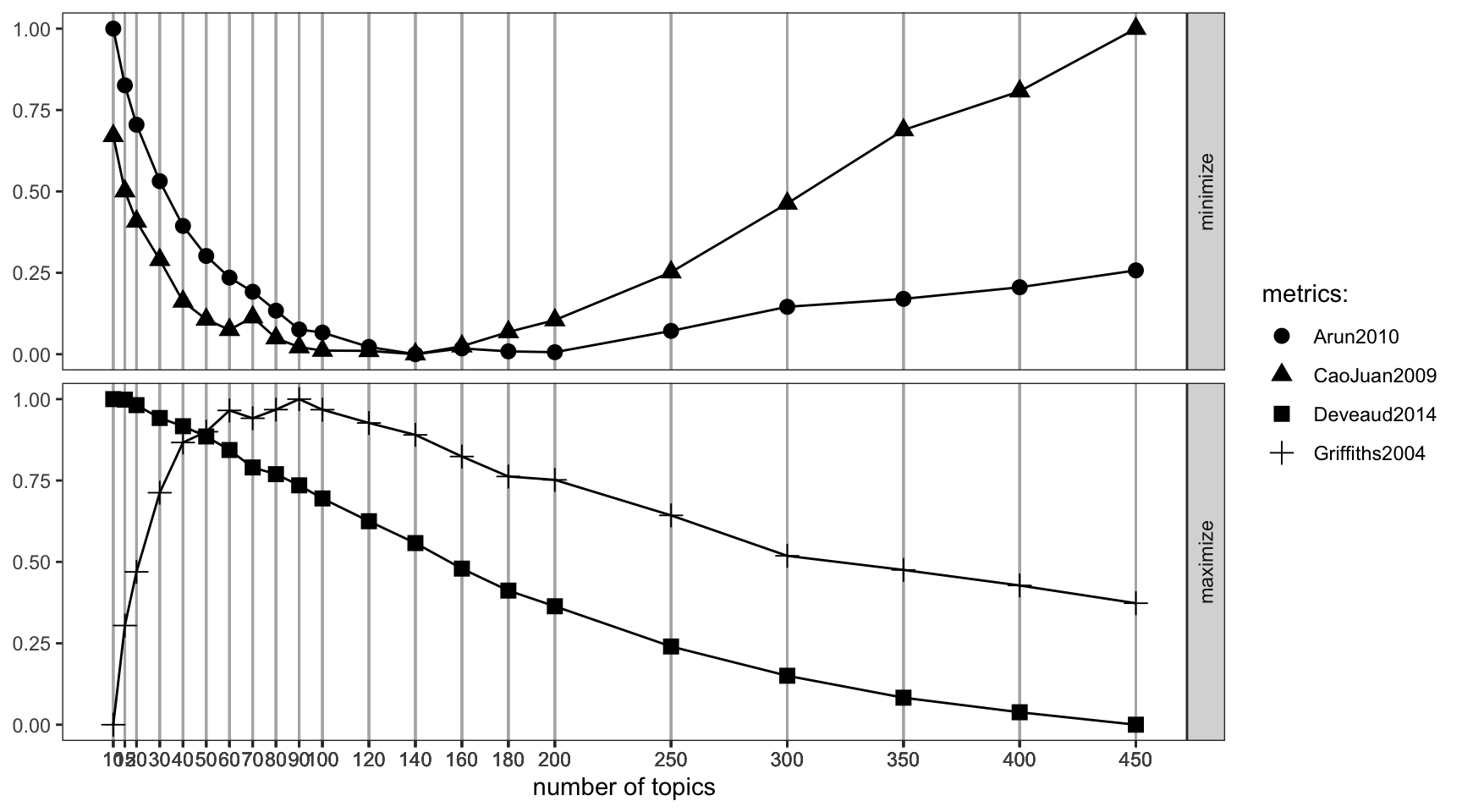
* minimization:
  + Arun2010 [1]
  + CaoJuan2009 [2]
* maximization:
  + Deveaud2014 [3]
  + Griffiths2004 [4,5]

Support function FindTopicsNumber\_plot can be used for easy analyze of the results

FindTopicsNumber\_plot(result)



Results calculated on the whole dataset (about 10 hours on quad-core computer) look like



From this plot can be made conclusion that optimal number of topics is in range 90-140. Metric Deveaud2014 is not informative in this situation.

OptimalNumber <- FindTopicsNumber(LDA\_xmas <- xmas.dfm.tfidf\_300 %>%

convert(to = "topicmodels"),

topics = seq(2, 8, by = 1),

mc.cores = 2,

metrics = c("CaoJuan2009", "Arun2010", "Deveaud2014"),

method = "VEM",

verbose = T)

FindTopicsNumber\_plot(OptimalNumber)

Therefore, we will choose 8 as the optimal number of topics.

LDA\_xmas <- xmas.dfm.tfidf\_300 %>%

convert(to = "topicmodels") %>%

LDA(k = 8)

We may use the package **tidytext** to inspect the topic probability distribution of each document, i.e. for each document the sum of the probabilities that it belongs to a topic from 1 to 8 is equal to 1.

LDA\_xmas %>%

tidy(matrix = "gamma") %>%

datatable(rownames = F)

Analogously, we can also obtain the probability distribution of words for each topic, i.e. for each topic the sum of probabilities that it generates different words is equal to 1.

LDA\_xmas %>%

tidy(matrix = "beta") %>%

datatable(rownames = F)

The top terms for each topic are:

# LDA for the Christmas songs

terms(LDA\_xmas, 10)

***t-SNE***

t-SNE stands for t-Statistics Stochastic Neighborhood Embedding, which is a dimensionality reduction technique that is formulated to capture the local clustering structure of the original data points. It is non-linear and non-deterministic.

We have generally speaking data points with high dimensionality $$x\_1, …, x\_n \in \mathbb{R}^N$$ and would like to calculate its counterparts $$y\_1, …, y\_n \in \mathbb{R}^M$$ in a low dimensional space, i.e. where $M

First of all, we define the probability that $x\_i$ would pick $x\_j$ as its neighbor as $$p\_{j|i} = \frac{exp(-||x\_j – x\_i||^2/2\sigma\_i^2)}{\sum\_{k\neq i} exp(-||x\_k – x\_i||^2/2\sigma\_i^2)}$$, i.e. it is proportional to a Gaussian centered at $x\_i$ where the variance $\sigma\_i$ is determined by a binary search such that the perplexity $$Perp(p\_i) = 2^{H(p\_i)} = 2^{-\sum\_{j\neq i} p\_{j|i}log\_2p\_{j|i}}$$ is as close as possible to a perplexity which is predefined by the user.

However, the conditional probability is not symmetric. In order to measure the similarity between $x\_i$ and $x\_j$, we define the metric to be $$p\_{ij} = \frac{p\_{j|i} + p\_{i|j}}{2}$$.

The similarity metric for $y\_1, …, y\_n$ is defined as the Student-t distribution with one-degree of freedom, i.e. the similarity between $y\_i$ and $y\_j$ is $$q\_{ij} = \frac{(1 + ||y\_j – y\_i||^2)^{-1}}{\sum\_{k \neq i}(1 + ||y\_k – y\_i||^2)^{-1}}$$.

The goal of t-SNE is to find the counterparts $\mathfrak{Y} = (y\_1, …, y\_n)$ of $\mathfrak{X} = (x\_1, …, x\_n)$ such that the Kullback-Leibler divergence $$D\_{KL}(P||Q) = \sum\_{i \neq j} p\_{ij}log\frac{p\_{ij}}{q\_{ij}}$$, i.e. our loss function $C$, which can be understood as the information loss using $\mathfrak{Y}$ to represent $\mathfrak{X}$, is minimized.

Obviously, there will be a relatively high loss if we use far apart pair $(y\_i, y\_j)$ to represent nearby pair $(x\_i, x\_j)$. Therefore, the local clustering/neighborhood structure of $\mathfrak{X}$ is preserved.

It can be shown that the gradient of the loss function has a relatively simple form of $$\frac{dC}{d\mathfrak{Y}} = (\frac{\partial C}{\partial y\_1}, …, \frac{\partial C}{\partial y\_n}) $$ where $$\frac{\partial C}{\partial y\_i} = 4\sum\_j (p\_{ij} – q\_{ij})(y\_i – y\_j)(1 + ||y\_i – y\_j||^2)^{-1}$$. The gradient descent is applied to minimize the loss function: $$\mathfrak{Y}^{(t)} = \mathfrak{Y}^{(t – 1)} + \eta\frac{dC}{d\mathfrak{Y}} + \alpha (t)(\mathfrak{Y}^{(t-1)} – \mathfrak{Y}^{(t – 2)})$$, where $\eta$ is called the learning rate and $\alpha(t)$ the momentum. $\mathfrak{Y}^{(0)}$ is a sample from an isotropic Gaussian with small variance.

The following computation will take about 30 minutes.

# t-Statistics Stochastic Neighbor Embedding --------------------------------

index.unique.songs <- !songs.dfm.tfidf\_300 %>%

as.matrix() %>%

duplicated()

songs.unique <- songs.dfm.tfidf\_300[index.unique.songs, ] %>% as.matrix()

tsne.all <- Rtsne(songs.unique)

songs\_2d <- tsne.all$Y %>%

as.data.frame() %>%

mutate(Label = label[index.unique.songs])

songs\_2d %>%

ggplot(aes(x = V1, y = V2, color = Label)) +

geom\_point(size = 0.25) +

scale\_color\_manual(values = c("Not Christmas" = "#a6a6a6",

"Christmas" = "#88ab33",

"Hidden Christmas" = "#F98948",

"Hidden Not Christmas" = "#437F97")) +

guides(color = guide\_legend(override.aes = list(size = 5))) +

theme(panel.grid.major = element\_blank(),

panel.grid.minor = element\_blank(),

panel.background = element\_rect(fill = "black"))

**What if we repeat the procedure for more than one iteration?**

So far we have only run the Naive Bayes for one iteration. However, we may repeat this procedure for more than one iteration, i.e. train a Naive Bayes classifier and relabel all the false positives as *Hidden Christmas*/*Christmas* and all the false negatives as *Hidden Not Christmas*/*Not Christmas* over and over again.

First of all, we prepare the data again to avoid bugs.

songs <- read.csv("songdata.csv")

songs$song <- songs$song %>% as.character()

songs$artist <- songs$artist %>% as.character()

songs$link <- songs$link %>% as.character()

songs$text <- songs$text %>% as.character()

# Initialization of the Labels

label <- character(dim(songs)[1])

for(i in 1:dim(songs)[1]){

if(str\_detect(songs$song[i], "Christmas") |

str\_detect(songs$song[i], "X-mas") |

str\_detect(songs$song[i], "Xmas")){

label[i] <- "Christmas"

} else{

label[i] <- "Not Christmas"

}

}

songs <- songs %>%

mutate(Label = label)

songs.dfm.tfidf <- corpus(songs, text\_field = "text",

docid\_field = "song") %>%

dfm(tolower = T,

stem = TRUE,

remove\_punct = TRUE,

remove = stopwords("english")) %>%

dfm\_trim(min\_count = 5, min\_docfreq = 3) %>%

dfm\_weight(type = "tfidf")

results <- data.frame(precision = numeric(10),

recall = numeric(10),

f1\_score = numeric(10))

Run 10 iterations.

for(i in 1:10){

# Determine the Indizes

christmas.index <- which(label == "Christmas")

not\_christmas.index <- which(label == "Not Christmas")

if(length(christmas.index) < length(not\_christmas.index)){

christmas.train.index <- christmas.index

not\_christmas.train.index <- sample(not\_christmas.index, length(christmas.index))

} else{

not\_christmas.train.index <- not\_christmas.index

christmas.train.index <- sample(christmas.index, length(not\_christmas.index))

}

train.index <- c(christmas.train.index, not\_christmas.train.index)

label.train <- label[train.index]

trainning.set <- songs.dfm.tfidf[train.index, ]

# Train the Model

classifier\_NB <- textmodel\_NB(trainning.set, label.train)

# Prediction

predictions <- classifier\_NB %>%

predict(newdata = songs.dfm.tfidf)

# Confusion Matrix

confusion <- table(predictions$nb.predicted, label)

precision <- confusion[1, 1]/sum(confusion[1, ])

recall <- confusion[1, 1]/sum(confusion[, 1])

f1\_score <- 2\*precision\*recall/(precision + recall)

# The hidden (not) Christmas Songs ----------------------------------------------

hidden.index <- (predictions$nb.predicted == "Christmas") & (songs$Label == "Not Christmas")

hidden\_not.index <- (predictions$nb.predicted == "Not Christmas") & (songs$Label == "Christmas")

hidden.xmas <- songs[hidden.index, ]

hidden.not\_xmas <- songs[hidden\_not.index, ]

label[hidden.index] <- "Hidden Christmas"

label[hidden\_not.index] <- "Hidden Not Christmas"

songs\_2d <- tsne.all$Y %>%

as.data.frame() %>%

mutate(Label = label[index.unique.songs])

random.forest <- songs\_2d %>%

ggplot(aes(x = V1, y = V2, color = Label)) +

geom\_point(size = 0.25) +

scale\_color\_manual(values = c("Not Christmas" = "#a6a6a6",

"Christmas" = "#88ab33",

"Hidden Christmas" = "#F98948",

"Hidden Not Christmas" = "#437F97")) +

guides(color = guide\_legend(override.aes = list(size = 5))) +

theme(panel.grid.major = element\_blank(),

panel.grid.minor = element\_blank(),

panel.background = element\_rect(fill = "black")) +

ggtitle(paste("Iteration:", i))

# Change the labels

label[hidden.index] <- "Christmas"

label[hidden\_not.index] <- "Not Christmas"

songs$Label <- label

songs.dfm.tfidf@docvars$Label <- label

results[i, ] <- c(precision, recall, f1\_score)

plot(random.forest)

}

results %>%

mutate(index = 1:10) %>%

melt(id = "index") %>%

ggplot(aes(x = index, y = value, color = variable)) +

geom\_line()

Then the precision as well as the f1 score grow monotonically at first and then converge to a value around 0.95, which means there are not many „Hidden Christmas Songs“ and „Hidden Not Christmas Songs“ left to be detected. However, in this procedure we always believe that the Naive Bayes classifier is 100% accurate, which is hardly possible. Thus, in each iteration there are some songs falsely classified by Naive Bayes as „Christmas“, which will be used in the next iteration in the training set to train the Naive Bayes classifier. With this accumulating error we might have the apprehension that the results are actually worse with more iterations.

At the end we have roughly half of the songs classified as „Christmas“ and the other half as „Not Christmas“, which seems very implausible. It raises the question whether or not there is an optimal number of iterations, however, we simply can not manually control whether all the 57,650 songs are correctly classified or not. This remains an open question to be answered.