Scribe Notes: Aug 9, Part 2

Christina Graves, Hitesh Prabhu

August 9, 2016

Text Analytics part 2

Recap and Scope of session

In the first session, we looked at the some of the foundational concepts of text analytics -

- How to represent documents
- Bag of words
- Data structures to store the bag of words
- Document-term matrix
- Tokenization.

In this session, we will be looking at

- TF-IDF weights
- Latent Semantic Indexing (LSI); (Also known as Latnet Semantic Analysis (LSA))

We will simulatenously be running you through an R script - `nyt_stories.R` - to illustrate some of the concepts discussed.

NOTE

1. The purpose of these scripts is to show how functions can be devised for text analysis. These type of functions carry out different tasks in the text pipeline process, such as tokenizing, stripping whitespace, stemming, removing singletons and computing bag of words into count vector. It is not recommended to create these type of functions on your own as they can become very tedious and complicated; instead text mining packages are available in R to perform text analysis.
2. Please note that all the functions used in `nyt_stories.R` are sourced from `textutils.R`.
3. For text analysis, *Scala* would be the ideal, scalable language to use. Twitter is built on *Scala*, and *Scala*'s syntax is similar to Python.

`nyt_stories.R` (till line number 27)

```
library(XML)

# Some helper functions
source('textutils.R')

# Read in the raw stories
art_stories = read.directory('../data/nyt_corpus/art/')
```

The above code loads the `XML` library (required for parsing XML documents) and reads all the XML documents in the directory.

```
# Turn these stories in vectors of counts of words
# also called a "bag of words"
art_stories_vec = LoW_to_countvector(art_stories)
```

```
# Pad out each vector to include NA's for words not seen in that story
# but part of the corpus vocabulary
art_stories_vec_std = standardize.ragged(art_stories_vec)
art_stories_vec_std = remove.singletons.ragged(art_stories_vec_std)

# Turn these lists into a document-term matrix
art_stories_DTM = make.BoW.frame(art_stories_vec_std)
```

- `LoW_to_countvector` essentially converts the bag of words into count vectors.
- `standardize.ragged` is used to make sure the index n corresponds to same word in all documents (makes them *NA*s if word not found in a document)
- `remove.singletons.ragged` removes singletons - which are words only seen once (this is optional and I just chose to do this)
- `make.BoW.frame` converts these lists into a document term matrix, while converting all *NA*s to 0.

TF-IDF Weighting

When creating a document term matrix for natural language processing, TF-IDF weighting is one method of calculating the entries of the matrix. TF-IDF weighting gives appropriate weights to the importance and frequency of terms in a document, calculating a score that is used as the entry for the matrix. The raw count of terms in a document is usually not used due to the varying lengths of documents. For example, a word that shows up 4 times in a 1000-word document is much less important that the same term appearing 4 times in a 10-word document.

If a word occurs very frequently in a document, it does not necessarily mean it is important (example: "a", "the", "and"). In this case, we would want to place less importance or weight on this term. Additionally, the word "tree" may be a meaningful distinction in one set of documents, but in a set of documents about classification trees, the frequency of "tree" wouldn't be very meaningful.

TF-IDF will give terms an appropriate weight based on their frequency. We want to downweight words that occur frequently across an entire corpus and we want to up weight very rare words. The TF component of TF-IDF will help with the up weighting and IDF will help with the down weighting.

TF = Term Frequency

IDF = Inverse Document Frequency

Recall the DTM:

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix}$$

Term Frequency (TF) Weight:

The number of a times a word appears in a document divided by the total number of words in a document. TF weight will be higher for words with higher frequencies.

N = Number of documents in a corpus

$N_i = \sum_{j=1}^{D} x_{ij}$

Inverse Document Frequency (IDF) Weight:

It takes the log of the number of documents in a corpus (N) divided by the number of documents containing a specific word. (M_j). As the ratio of (N/M_j) becomes closer to one, the IDF approaches zero. So, if every document contains the word j, then that word is given zero importance by the IDF function. This offsets the high frequency score given by TF function to very common words.

M_j = Number of documents containing word j

$$M_j = \begin{cases} 1, & \text{if } x_{ij} > 1 \\ 0, & \text{otherwise} \end{cases}$$

Defintion: TF-IDF Weight

The total TF-IDF weight is the product of the Term Frequency TF with the Inverse Document Frequency of a term. This score will be the entry in the document term matrix (DTM).

TF-IDF = TF(X_{ij}) * IDF(X_{ij})

Introduction to Latent Semantic Indexing (LSI)/Latent Semantic Analysis (LSA):

Key Idea of LSI/LSA: Matching similar documents with one another (perhaps for a recommendation engine)

Why can't we simply match documents that contain the same words?

Two problems: **Synonymy** *and* **Polysemy**

Synonymy: Two documents might be talking about same subject material but using different words to convey the same meaning. If matching is only based on the syntax of these documents, the similarity of these documents would be overlooked. It is important to also consider the semantics used within documents to detect similarities.

Polysemy: Opposite of the synonym problem, polysemy refers to words that have multiple meanings in different contexts. For example, a "tree" for a botanist may have a different meaning than "tree" for a data scientist. Focusing on only the syntax of these terms could lead to incorrectly matching documents.

Getting around these problems: **Context**

If two words are used in a similar context (used with similar collections of words) in different documents, then a similarity between the documents can be more accurately concluded rather than just looking at the presence of those words alone; latent semantic indexing helps to carry out this task.

The LSI method = Principal Components Analysis (PCA) of the document term matrix or its derivative.

How does this solve our problem?

- Every document will have weights associated with it; each ingredient will have something that conveys the meaning of that document.
- Every word will have a set of scores associated with it that will tell you which ingredients are heavily involved with those words

Recap We thought about PCA as looking at ingredients in a recipe. Here an ingredient is a vector that is of the same dimension as the number of features in DTM.

Ingredients/PC's ~ vectors whose indexes map to words

Weights/scores ~ vector whose indexes correspond to documents

If a word occurs in a systematic fashion in 2 or more documents, we can match these.