# Big Data Analytics

## Chapter 4: Text Processing with Latent Semantic Analysis

Following: [3] "**Advanced Analytics with Spark**", Chapter 6

Prof. Dr. Martin Theobald
**Faculty of Science, Technology & Communication**
**University of Luxembourg**

UNIVERSITÉ DU LUXEMBOURG

# How to Analyze Text Data?

Unlike the previous data sets, text data is often considered as "**unstructured data**" – although this is far from actually being true:

- In fact, natural language follows a very fine-grained structure, with a well-defined grammar and usually a clear intention of an author in describing a possibly complex matter. However, this structure still is quite difficult to process for a machine…

Thus, in the following we focus on a very **basic semantic analysis** of words occurring in documents based on **large-scale co-occurrence statistics**.

We will represent these statistics in the form of a **large** (and usually very **sparse**) **matrix**, where the rows denote words and the columns denote documents.

The cell entries of the matrix are usually based on **TF-IDF weights**:

- The **term frequency** (TF) of a word is the number of times the word (or its lemma) occurs within a document.
- The **document frequency** (DF) of a word is the number of documents that contain the word (or its lemma).
- This coincides with a so-called **bag-of-words representation** for documents in many Information Retrieval and Data Mining applications.

# The Wikipedia Data Set

- We have extracted 500 MB of **English Wikipedia articles** from a recent snapshot and converted the resulting 41,784 articles into a plain-text (UTF-8) format.

- The dump is split into 1,060 individual files, each of about 500 KB size.

- Each such split is of the following format:

```
<doc id="38298" url="https://en.wikipedia.org/wiki?curid=38298"
        title="George Boole">
         George Boole was an English mathematician, educator, philosopher
        and logician. ...
<doc>
```

# 4.1 Latent Semantic Analysis

# Latent Semantic Analysis vs. Latent Semantic Indexing

**Latent Semantic Analysis** (LSA) and **-Indexing** (LSI) are mostly synonymous.

- The general technique of applying a Singular Value Decomposition (SVD) to a **word-document matrix** $\boldsymbol{A}$ is called Latent Semantic Analysis (LSA).
- By applying SVD to $\boldsymbol{A}$, similar words are mapped to their **implicit** (i.e., "latent") **meanings**, which are represented as a $k$-dimensional vector for each word.
- Conversely, documents are also represented by $k$-dimensional vectors, which each represent the strength of the **latent concepts** the documents contains.
- The usage of LSA for **indexing documents** in the context of **Information Retrieval** and/or **Data Mining** is usually called Latent Semantic Indexing.

**Basic LSI Idea**:

- When given the following documents:

  $\boldsymbol{d}_1 = "Jaguar\ animal\ cat"$

  $\boldsymbol{d}_2 = "Jaguar\ car\ brand"$

  $\boldsymbol{d}_3 = "Jaguar\ luxury\ vehicle"$

- The query $\boldsymbol{q} = "car\ brand"$ should return $\boldsymbol{d}_2$, $\boldsymbol{d}_3$, $\boldsymbol{d}_1$ (in that particular order).

# Singular Value Decomposition

Generally, an **SVD of a word-document matrix** $\boldsymbol{A}$ is of the following form:

$$\boldsymbol{A}_{m\times n} = \boldsymbol{U}_{m\times m}\,\boldsymbol{S}_{m\times n}\,\boldsymbol{V}^T{}_{n\times n}$$

- $\boldsymbol{U}$ is an orthogonal matrix (i.e., $\boldsymbol{U}\,\boldsymbol{U}^T = \boldsymbol{I}$) whose columns are orthonormal eigenvectors of $\boldsymbol{A}\,\boldsymbol{A}^T$.
- $\boldsymbol{S}$ is a diagonal matrix containing the square roots of the eigenvalues of $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively, *in descending order*.
- $\boldsymbol{V}^T$ is the transpose of an orthogonal matrix (i.e., $\boldsymbol{V}\,\boldsymbol{V}^T = \boldsymbol{I}$) whose rows are orthonormal eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$.

Suppose that $\boldsymbol{A}$ is an $m \times n$ matrix, then a **full SVD decomposition** yields an $m \times m$ matrix $\boldsymbol{U}$, an $m \times n$ matrix $\boldsymbol{S}$ and an $n \times n$ matrix $\boldsymbol{V}^T$.

However, by **reducing $\boldsymbol{S}$** to its $k$ largest singular values, we may achieve the desired **dimensionality reduction** also for $\boldsymbol{U}$ and $\boldsymbol{V}^T$:

$$\boldsymbol{A}_{m\times n} \approx \boldsymbol{U}_{m\times k}\,\boldsymbol{S}_{k\times k}\,\boldsymbol{V}^T{}_{k\times n}$$

The goal here usually is not to reconstruct $\boldsymbol{A}$, but to find word-to-word or document-to-document similarities:

- **Word-to-word similarities**: compare rows $\boldsymbol{U}_{m\times k}\,\boldsymbol{S}_{k\times k}$
- **Document-to-document similarities**: compare rows $[\boldsymbol{S}_{k\times k}\,\boldsymbol{V}^T{}_{k\times n}]^T = \boldsymbol{V}_{n\times k}\,\boldsymbol{S}_{k\times k}$

# SVD Example: Full Decomposition (I)

$$\boldsymbol{A} = \begin{bmatrix} 2 & 0 & 8 & 6 & 0 \\ 1 & 6 & 0 & 1 & 7 \\ 5 & 0 & 7 & 4 & 0 \\ 7 & 0 & 8 & 5 & 0 \\ 0 & 10 & 0 & 0 & 7 \end{bmatrix} = \begin{bmatrix} -0.54 & 0.07 & 0.82 & -0.11 & 0.12 \\ -0.10 & -0.59 & -0.11 & -0.79 & -0.06 \\ -0.53 & 0.06 & -0.21 & 0.12 & -0.81 \\ -0.65 & 0.07 & -0.51 & 0.06 & 0.56 \\ -0.06 & -0.80 & 0.09 & 0.59 & 0.04 \end{bmatrix}$$

$$\times \begin{bmatrix} 17.92 & 0 & 0 & 0 & 0 \\ 0 & 15.17 & 0 & 0 & 0 \\ 0 & 0 & 3.56 & 0 & 0 \\ 0 & 0 & 0 & 1.98 & 0 \\ 0 & 0 & 0 & 0 & 0.35 \end{bmatrix}$$

$$\times \begin{bmatrix} -0.46 & 0.02 & -0.87 & 0.00 & 0.17 \\ -0.07 & -0.76 & 0.06 & 0.60 & 0.23 \\ -0.74 & 0.10 & 0.28 & 0.22 & -0.56 \\ -0.48 & 0.03 & 0.40 & -0.33 & 0.70 \\ -0.07 & -0.64 & -0.04 & -0.69 & -0.32 \end{bmatrix}$$

- This full matrix decomposition is indeed **lossless**, i.e., the product $\boldsymbol{U}_{m\times m}\,\boldsymbol{S}_{m\times n}\,\boldsymbol{V}^T{}_{n\times n}$ returns again $\boldsymbol{A}_{m\times n}$.

# SVD Example: Full Decomposition (II)

- Computing **word-to-word similarities** based on

$$\boldsymbol{W}_{m\times n} = \boldsymbol{U}_{m\times m}\ \boldsymbol{S}_{m\times n} = \begin{bmatrix} -9.72 & 0.99 & 2.93 & -0.21 & 0.04 \\ -1.82 & -9.00 & -0.40 & -1.56 & -0.02 \\ -9.41 & 0.90 & -0.76 & 0.23 & -0.28 \\ -11.56 & 1.07 & -1.81 & 0.12 & 0.20 \\ -1.16 & -12.09 & 0.32 & 1.18 & 0.02 \end{bmatrix}$$

- and using, e.g., **Cosine similarity**

$$\mathrm{CosSim}(\boldsymbol{w}_i, \boldsymbol{w}_j) = \frac{\boldsymbol{w}_i \cdot \boldsymbol{w}_j}{||\boldsymbol{w}_i||_2 ||\boldsymbol{w}_j||_2}$$

- we get:

$$\begin{aligned} \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_1) &= 1.00 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_2) &= 0.08 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_3) &= 0.93 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_4) &= 0.90 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_5) &= 0.00 \end{aligned}$$

- These values coincide with the pairwise row distances in $\boldsymbol{A}$.

# SVD Example: Full Decomposition (III)

- Computing **document-to-document similarities** based on

$$
\boldsymbol{D}_{n\times m} = [\boldsymbol{S}_{m\times n}\, \boldsymbol{V}^T{}_{n\times n}]^T = \begin{bmatrix} -8.33 & -0.33 & -3.10 & -0.00 & 0.06 \\ -1.26 & -11.53 & 0.22 & 1.19 & 0.08 \\ -13.17 & 1.50 & 1.01 & 0.44 & -0.20 \\ -8.68 & 0.39 & 1.42 & -0.66 & 0.25 \\ -1.16 & -9.73 & -0.16 & -1.37 & -0.11 \end{bmatrix}
$$

- and again using **Cosine similarity**

$$
\mathrm{CosSim}(\boldsymbol{d}_i, \boldsymbol{d}_j) = \frac{\boldsymbol{d}_i \cdot \boldsymbol{d}_j}{||\boldsymbol{d}_i||_2 ||\boldsymbol{d}_j||_2}
$$

- we get:

$$
\begin{aligned}
\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_1) &= 1.00 \\
\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_2) &= 0.06 \\
\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_3) &= 0.90 \\
\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_4) &= 0.87 \\
\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_5) &= 0.08
\end{aligned}
$$

- These values in turn coincide with the pairwise column distances in $\boldsymbol{A}$.

# SVD Example: Reduced Decomposition (I)

$k = 3$

$$\boldsymbol{A} = \begin{bmatrix} 2 & 0 & 8 & 6 & 0 \\ 1 & 6 & 0 & 1 & 7 \\ 5 & 0 & 7 & 4 & 0 \\ 7 & 0 & 8 & 5 & 0 \\ 0 & 10 & 0 & 0 & 7 \end{bmatrix} \approx \left[\begin{array}{ccc|cc} -0.54 & 0.07 & 0.82 & -0.11 & 0.12 \\ -0.10 & -0.59 & -0.11 & -0.79 & -0.06 \\ -0.53 & 0.06 & -0.21 & 0.12 & -0.81 \\ -0.65 & 0.07 & -0.51 & 0.06 & 0.56 \\ -0.06 & -0.80 & 0.09 & 0.59 & 0.04 \end{array}\right]$$

$$\times \begin{bmatrix} 17.92 & 0 & 0 & 0 & 0 \\ 0 & 15.17 & 0 & 0 & 0 \\ 0 & 0 & 3.56 & 0 & 0 \\ 0 & 0 & 0 & 1.98 & 0 \\ 0 & 0 & 0 & 0 & 0.35 \end{bmatrix}$$

$$\times \left[\begin{array}{ccccc} -0.46 & 0.02 & -0.87 & 0.00 & 0.17 \\ -0.07 & -0.76 & 0.06 & 0.60 & 0.23 \\ -0.74 & 0.10 & 0.28 & 0.22 & -0.56 \\ \hline -0.48 & 0.03 & 0.40 & -0.33 & 0.70 \\ -0.07 & -0.64 & -0.04 & -0.69 & -0.32 \end{array}\right]$$

- This reduced matrix decomposition now becomes "**lossy**", i.e., the product $\boldsymbol{U}_{m\times k}\, \boldsymbol{S}_{k\times k}\, \boldsymbol{V}^{T}{}_{k\times n}$ returns only an approximation of $\boldsymbol{A}_{m\times n}$.

# SVD Example: Reduced Decomposition (II)

- Computing **word-to-word similarities** based on

$k = 3$

$$\boldsymbol{W}_{m\times k} = \boldsymbol{U}_{m\times k}\ \boldsymbol{S}_{k\times k} = \begin{bmatrix} -9.72 & 0.99 & 2.93 \\ -1.82 & -9.00 & -0.40 \\ -9.41 & 0.90 & -0.76 \\ -11.56 & 1.07 & -1.81 \\ -1.16 & -12.09 & 0.32 \end{bmatrix}$$

- and using, e.g., **Cosine similarity**

$$\mathrm{CosSim}(w_i, w_j) = \frac{\boldsymbol{w}_i \cdot \boldsymbol{w}_j}{\left|\left|\boldsymbol{w}_i\right|\right|_2 \left|\left|\boldsymbol{w}_j\right|\right|_2}$$

- we get:

$$\begin{aligned} \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_1) &= 1.00 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_2) &= 0.08 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_3) &= 0.93 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_4) &= 0.90 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_5) &= 0.00 \end{aligned}$$

- These values now approximate the pairwise row distances in $\boldsymbol{A}$.

# SVD Example: Reduced Decomposition (III)

$k = 3$

- Computing **document-to-document similarities** based on

$$\boldsymbol{D}_{n\times k} = [\boldsymbol{S}_{k\times k}\, \boldsymbol{V}^T{}_{k\times n}]^T = \begin{bmatrix} -8.33 & 1.06 & -2.62 \\ 0.39 & -11.53 & 0.35 \\ -15.56 & 0.96 & 1.01 \\ -0.01 & 9.12 & 0.80 \\ 3.07 & 3.46 & -2.01 \end{bmatrix}$$

- and again using **Cosine similarity**

$$\mathrm{CosSim}(\boldsymbol{d}_i, \boldsymbol{d}_j) = \frac{\boldsymbol{d}_i \cdot \boldsymbol{d}_j}{\|\boldsymbol{d}_i\|_2 \|\boldsymbol{d}_j\|_2}$$

- we get:

$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_1) = 1.00$$
$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_2) = 0.08$$
$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_3) = 0.92$$
$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_4) = -0.14$$
$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_5) = -0.54$$

- These values now approximate the pairwise column distances in $\boldsymbol{A}$.

# Processing Queries

- A **new query vector** $\boldsymbol{q}_{1\times m}$ can be processed against the reduced decomposition by transforming it in a similar manner, namely by computing

$$\boldsymbol{q}'_{1\times k} = \boldsymbol{q}_{1\times m} \times \boldsymbol{U}_{m\times k} \times \boldsymbol{S}^{-1}_{k\times k}$$

- and then by computing the pairwise Cosine similarities of $\boldsymbol{q}_{1\times k}$ with the row vectors in $\boldsymbol{D}_{n\times k}$ which each represents a reduced document vector.

<u>Example:</u>

- For the query vector $\boldsymbol{q}_{1\times m} = [1,3,0,2,0]$, we obtain the **transformed and reduced query vector** $\boldsymbol{q}'_{1\times k} = [-0.12, -0.10, -0.15]$ with the following Cosine similarities to the row vectors in $\boldsymbol{D}_{n\times k} = [\boldsymbol{S}_{k\times k}\, \boldsymbol{V}^{T}{}_{k\times n}]^{T}$ (see previous slide):

$$\begin{aligned}
\text{CosSim}(\boldsymbol{q}', \boldsymbol{d}_1) &= 0.78\\
\text{CosSim}(\boldsymbol{q}', \boldsymbol{d}_2) &= 0.44\\
\text{CosSim}(\boldsymbol{q}', \boldsymbol{d}_3) &= 0.47\\
\text{CosSim}(\boldsymbol{q}', \boldsymbol{d}_4) &= -0.53\\
\text{CosSim}(\boldsymbol{q}', \boldsymbol{d}_5) &= -0.39
\end{aligned}$$

$k = 3$

# Finding an SVD of a Word-Document Matrix (I)

That is, just like the matrix factorization shown in Chapter 3, we can compute the SVD of a matrix $\boldsymbol{A}$ by **solving a set of linear equations**:

1. Compute the $k$ largest (non-zero) eigenvalues of either $\boldsymbol{A}\,\boldsymbol{A}^T$ or $\boldsymbol{A}^T\boldsymbol{A}$.

2. To find $\boldsymbol{U}$:
   - Use these eigenvalues to compute $k$ eigenvectors of $\boldsymbol{A}\,\boldsymbol{A}^T$.
   - Turn each eigenvector into a column vector of a new matrix $\boldsymbol{U}'$.
   - Convert $\boldsymbol{U}'$ into the orthogonal matrix $\boldsymbol{U}$ (e.g., using the Gram–Schmidt procedure)

3. To find $\boldsymbol{V}^T$:
   - Use these eigenvalues to compute $k$ eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$.
   - Turn each eigenvector into a row vector of a new matrix $\boldsymbol{V}^{T\prime}$.
   - Convert $\boldsymbol{V}^{T\prime}$ into the orthogonal matrix $\boldsymbol{V}^T$ (e.g., using the Gram–Schmidt procedure)

4. To find $\boldsymbol{S}$:
   - Populate a new $k \times k$ diagonal matrix with the square roots of the non-zero eigenvalues computed in step 1.

# Finding an SVD of a Word-Document Matrix (II)

- Spark imports a number of low-level Fortran libraries for matrix operations (ARPACK) that provide different numerical optimizations also for **approximate eigenvalue decompositions**, including SVD.

- See also here for a **tutorial on LSA** via SVD and eigenvalue decomposition.

- See the previous chapter on possible ways to implement **distributed matrix multiplications** (Methods 1 & 2) in Spark.

# 4.2 Text Analysis & Singular-Value-Decomposition in Spark's MLlib

# Loading the Data Set (I)

- First of all, we need to define a special loading function for our new Wikipedia file format.
- Recall that a Wikipedia article is delimited by a pair of `<doc …>` and `</doc>` tags, such that a simple line-based input format would not guarantee that consecutive lines in the input documents are also stored as consecutive lines generated by the `sc.textFile(…)` RDD.
- On the other hand, we do not want to assume that all Wikipedia files fit into the main-memory of our driver node, hence we should keep working with partitioned RDDs from the beginning.
- Fortunately, there is an alternative function that loads whole text files into an RDD:

```
sc.wholeTextFiles("./pathToTextFiles")
```

# Loading the Data Set (II)

- We can then transform the RDD with the input files (each containing many Wikipedia articles) into a new RDD containing **one entry per Wikipedia article** by a single `flatMap` transformation:

```
val textFiles = sc.wholeTextFiles("./path-to-wiki-articles")
val plainText = textFiles.flatMap{ case (uri, text) =>
   parse(text.split("\n")) } // contains one entry for each
                                        Wikipedia article
```

- Note that the `parse` function (see Moodle) is executed on all partitions of input files **in parallel**.

- This way, we do **not need to assume that all articles fit into the main memory of our driver node** at any time. Every processing step is performed only via RDD transformations!

# Create a Default NLP Pipeline

- Our basic **natural-language-processing** (NLP) **pipeline** consists of:
  - Sentence splitting
  - Tokenization
  - Part-Of-Speech (POS) tagging
  - Lemmatization

- The **Stanford CoreNLP tools** provide a very comprehensive API for these (and other) NLP tasks:

```
import edu.stanford.nlp.pipeline._
import edu.stanford.nlp.ling.CoreAnnotations._

def createNLPPipeline(): StanfordCoreNLP = {
  val props = new Properties()
  props.put("annotators", "tokenize, ssplit, pos, lemma")
  new StanfordCoreNLP(props)
}
```

# Filtering Out Non-Letter Tokens & Stopwords

- For a more compact dictionary of words, we may still want to filter out tokens that are **not proper sequences of letters** and very frequent words, the so-called **stopwords**.

```
def isOnlyLetters(str: String): Boolean = {
  str.forall(c => Character.isLetter(c)) }
```

- Stopwords are extremely frequent words of a language, such as **prepositions**, **articles**, **auxiliary verbs**, etc., that do not add much information to our semantic indexing approach.

```
import scala.io.Source._
val bStopWords = sc.broadcast(
  fromFile("stopwords.txt").getLines().toSet)
```

- A respective file with English stopwords is available on Moodle.

# From Text to Word Lemmas

- An input text of type `String` should be tokenized and the words should be reduced to their lemmatized form:

```
def plainTextToLemmas(text: String,
    pipeline: StanfordCoreNLP): Seq[String] = {
  val doc = new Annotation(text)
  pipeline.annotate(doc)
  val lemmas = new ArrayBuffer[String]()
  val sentences = doc.get(classOf[SentencesAnnotation])
  for (sentence <- sentences;
      token <- sentence.get(classOf[TokensAnnotation])) {
    val lemma = token.get(classOf[LemmaAnnotation])
    if (lemma.length > 2 && !bStopWords.value.contains(lemma)
        && isOnlyLetters(lemma)) {
      lemmas += lemma.toLowerCase } }
  lemmas }
```

# Initialize the NLP Pipelines

- Use a `mapPartitions` transformation on the `plainText` RDD so that we only **initialize** the NLP pipeline object **once per partition** instead of once per document:

```
val lemmatized: RDD[Seq[String]] =
    plainText.mapPartitions(it => {
  val pipeline = createNLPPipeline()
  it.map { case(title, contents) =>
    plainTextToLemmas(contents, pipeline)
  }
})
```

# Computing TF Weights

- The term frequency of a term in a document is a local property of that document. So we can compute it via a **local aggregation** over the documents.

```
import scala.collection.mutable.HashMap

val docTermFreqs = lemmatized.map(terms => {
  val termFreqs = terms.foldLeft(new HashMap[String, Int]()) {
    (map, term) => {
      map += term -> (map.getOrElse(term, 0) + 1)
    map }
  }
  termFreqs
})
```

- For frequent access, the **term-frequency map** should be **cached**.

```
docTermFreqs.cache()
```

# Remove Infrequent Terms

- To further reduce the term space, we may run a **standard word count** to filter out infrequent terms (in this case with a document frequency of less than 12):

```
val docFreqs = docTermFreqs.flatMap(_.keySet).map((_, 1)).
  reduceByKey(_ + _, 12)
```

- Keep only the **top 50,000 terms** from within all documents:

```
val ordering = Ordering.by[(String, Int), Int](_._2)
val topDocFreqs = docFreqs.top(50000)(ordering)
```

# Computing DF Weights (I)

The document frequency of a term across all documents is a global property of the collection. Since we are operating over the partitioned data structure `docTermFreqs`, we need to aggregate it in two steps:

1. We count the **document frequencies locally for each partition**.

```
val zero = new HashMap[String, Int]()

def merge(dfs: HashMap[String, Int],
          tfs: (String, HashMap[String, Int]))
    : HashMap[String, Int] = {
  tfs._2.keySet.foreach { term =>
    dfs += term -> (dfs.getOrElse(term, 0) + 1) }
  dfs
}
```

# Computing DF Weights (II)

The document frequency of a term across all documents is a global property of the collection. Since we are operating over the partitioned data structure `docTermFreqs`, we need to aggregate it in two steps:

2. We combine the **document frequencies globally across all partitions**.

```
def comb(dfs1: HashMap[String, Int], dfs2: HashMap[String, Int])
    : HashMap[String, Int] = {
  for ((term, count) <- dfs2) {
    dfs1 += term -> (dfs1.getOrElse(term, 0) + count) }
  dfs1
}
```

# Computing DF Weights (III)

The document frequency of a term across all documents is a global property of the collection. Since we are operating over the partitioned data structure `docTermFreqs`, we need to aggregate it in two steps:

```
docTermFreqs.aggregate(zero)(merge, comb)
```

And finally invert the DF values into their **IDF counterparts**:

```
val idfs = topDocFreqs.map {
  case (term, count) =>
    (term, math.log(numDocs.toDouble / count))
}.toMap
```

And **broadcast** this map:

```
bIdfs = sc.broadcast(idfs)
```

# Broadcast the Term Dictionary

- Do not forget to **broadcast the static term-id dictionary**:

```
val termIds = idfs.keys.zipWithIndex.toMap
val bTermIds = sc.broadcast(termIds)
```

# Finally Merge the TF and IDF Weights into Sparse Vectors

- The last transformation **combines the TF and IDF weights** for all terms in a document into a **sparse vector** representation:

```
import scala.collection.JavaConversions._
import org.apache.spark.mllib.linalg.Vectors

val vecs = docTermFreqs.map(termFreqs => {
  val docTotalTerms = termFreqs.values().sum
  val termScores = termFreqs.filter {
    case (term, freq) => bTermIds.value.containsKey(term)
  }.map{
    case (term, freq) => (bTermIds.value(term),
      bIdfs.value(term) * termFreqs(term) / docTotalTerms)
  }.toSeq
  Vectors.sparse(bTermIds.value.size, termScores)
})
```

# Compute the Singular Value Decomposition

- The `RowMatrix` class is used to represent entries of a **document-term matrix** in a sparse way. It is directly used as input for the SVD computation:

```
import org.apache.spark.mllib.linalg.distributed.RowMatrix

vecs.persist()
val mat = new RowMatrix(vecs)
val k = 1000 // number of latent concepts
                   in the reduced matrix
val svd = mat.computeSVD(k, computeU=true)
```

- Caution: Spark MLlib *unfortunately* swaps the common convention used to encode the term-document matrix $\boldsymbol{A}$:
  - `svd.V` encodes the word-to-concept mappings
  - `svd.U` encodes the document-to-concept mappings
  - `svd.S` contains the singular values of $\boldsymbol{A}$ (in descending order)

# Finding the Top Terms for Latent Concepts

We can now use $V$ to inspect the **latent concepts** represented in our Wikipedia collection.

- First, we may want to **rank terms** by their similarity to the top concepts:

```
def topTermsInTopConcepts(
      svd: SingularValueDecomposition[RowMatrix, Matrix],
      numConcepts: Int, numTerms: Int): Seq[Seq[(String, Double)]] = {
    val v = svd.V
    val topTerms = new ArrayBuffer[Seq[(String, Double)]]()
    for (i <- 0 until numConcepts) {
      val offs = i * v.numRows
      val termWeights = v.toArray.slice(offs, offs + v.numRows).zipWithIndex
      val sorted = termWeights.sortBy(-_._1)
      topTerms += sorted.take(numTerms).map {
        case (score, id) =>
          (bTermIds.value.find(_._2 == id).getOrElse(("", -1))._1, score) } }
    topTerms }
```

# Finding the Top Documents for the Latent Concepts

We can also use $\boldsymbol{U}$ to inspect the **latent concepts** represented in our Wikipedia collection.

- Next, we may want to **rank documents** by these top concepts:

```
def topDocsInTopConcepts(
   svd: SingularValueDecomposition[RowMatrix, Matrix],
   numConcepts: Int, numDocs: Int): Seq[Seq[(Long, Double)]] = {
   val u = svd.U
   val topDocs = new ArrayBuffer[Seq[(Long, Double)]]()
   for (i <- 0 until numConcepts) {
     val docWeights = u.rows.map(_.toArray(i)).zipWithUniqueId()
     topDocs += docWeights.top(numDocs).map {
       case (score, id) => (id, score) } }
   topDocs }
```

# Print the Results

- Now we may finally **print the results of all our efforts** so far by using the two afore defined functions:

```
val topConceptTerms = topTermsInTopConcepts(svd, 4, 10)
val topConceptDocs = topDocsInTopConcepts(svd, 4, 10)
for ((terms, docs) <- topConceptTerms.zip(topConceptDocs)) {
  println("Concept terms: " + terms.map(_._1).mkString(", "))
  println("Concept docs: " + docs.map(_._1).mkString(", "))
  println()
}
```

- Note that MLlib stores $\boldsymbol{V}$ **locally at the driver node**, while $\boldsymbol{U}$ is stored in a **distributed manner across all executor nodes** (being sharded on its rows).

# Querying the Latent Semantic Index

With the reduced decomposition of $\boldsymbol{A}$ into $\boldsymbol{U} \times \boldsymbol{S} \times \boldsymbol{V}^T$ at hand, we can now perform the following tasks in the "latent" concept space instead of using the actual terms and documents:

- **Document-to-document similarities**:
  - Compute row similarities of $\boldsymbol{U} \times \boldsymbol{S}$
- **Term-to-term similarities**:
  - Compute row similarities of $\boldsymbol{V} \times \boldsymbol{S}$
- **Single-term-to-document similarities:**
  - Multiply $\boldsymbol{U} \times \boldsymbol{S}$ with the row vector of term $i$ in $\boldsymbol{V}$
- **Multi-term-to-document similarities:**
  - Transform $\boldsymbol{q}' = \boldsymbol{q}_{1 \times m} \times \boldsymbol{V}_{m \times k}$
  - Multiply $\boldsymbol{U} \times \boldsymbol{S}$ with the transformed query vector $\boldsymbol{q}'^T$

# Document-Document Relevance

- Based on the reduced representation, we can find the **most similar documents** to a given query document:

```
import org.apache.spark.mllib.linalg.Matrices

def topDocsForDoc(normalizedUS: RowMatrix, docId: Long)
    : Seq[(Double, Long)] = {
  val docRowArr = row(normalizedUS, docId)
  val docRowVec = Matrices.dense(docRowArr.length, 1, docRowArr)
  val docScores = normalizedUS.multiply(docRowVec)
  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
  allDocWeights.filter(!_._1.isNaN).top(10)
}
val US = multiplyByDiagonalMatrix(svd.U, svd.s)
val normalizedUS = rowsNormalized(US)
topDocsForDoc(normalizedUS, idDocs(doc), docIds)
```

# Term-Term Relevance

- Similarly, we can also find the **top similar terms** to a given query term:

```
import breeze.linalg.{SparseVector => BSparseVector}
import breeze.linalg.{DenseVector => BDenseVector}
import breeze.linalg.{DenseMatrix => BDenseMatrix}

def topTermsForTerm(
    normalizedVS: BDenseMatrix[Double],
    termId: Int): Seq[(Double, Int)] = {
  val rowVec = new BDenseVector[Double](
    row(normalizedVS, termId).toArray)
  val termScores = (normalizedVS * rowVec).toArray.zipWithIndex
  termScores.sortBy(-_._1).take(10)
}

val VS = multiplyByDiagonalMatrix(svd.V, svd.s)
val normalizedVS = rowsNormalized(VS)
topTermsForTerm(normalizedVS, idTerms(term), termIds)
```

# Term-Document Relevance

- As in an actual **Information Retrieval** setting, we can find the **most similar documents** for a given query term:

```
def topDocsForTerm(US: RowMatrix, V: Matrix, termId: Int)
    : Seq[(Double, Long)] = {
  val termRowArr = row(V, termId).toArray
  val termRowVec = Matrices.dense(termRowArr.length, 1, termRowArr)
  val docScores = US.multiply(termRowVec)
  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
  allDocWeights.top(10)
}

topDocsForTerm(normalizedUS, svd.V, idTerms(term))
```

# Multi-Term Queries (I)

- First, we transform a query vector into a compatible `SparseVector` representation using IDF values as term weights:

```
def termsToQueryVector(
    terms: Seq[String],
    idTerms: Map[String, Int],
    idfs: Map[String, Double]): BSparseVector[Double] = {
  val indices = terms.map(idTerms(_)).toArray
  val values = terms.map(idfs(_)).toArray
  new BSparseVector[Double](indices, values, idTerms.size)
}
```

# Multi-Term Queries (II)

- Finally, we multiply $\boldsymbol{U} \times \boldsymbol{S}$ with the transformed query vector $\boldsymbol{q}'^T$:

```
def topDocsForTermQuery(
    US: RowMatrix,
    V: Matrix,
    query: BSparseVector[Double]): Seq[(Double, Long)] = {
  val breezeV = new BDenseMatrix[Double](V.numRows, V.numCols, V.toArray)
  val termRowArr = (breezeV.t * query).toArray
  val termRowVec = Matrices.dense(termRowArr.length, 1, termRowArr)
  val docScores = US.multiply(termRowVec)
  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
  allDocWeights.top(10) }

val queryVec = termsToQueryVector(terms, idTerms, idfs)
topDocsForTermQuery(US, svd.V, queryVec)
```

# Summary

- **LSA** and related techniques, such as PCA, pLSA, LDA, are very broadly applied techniques for **analyzing text data**.
  - Dimensionality reduction
  - Content filtering
  - Clustering
  - Visualization
  - Indexing & retrieval

- **LSA** also has a variety of applications also **outside text analysis**:
  - Face detection ("eigenfaces") to detect common patterns in human appearance
  - Detection of climate patterns, etc.