Lab 7: Search and Retrieval

May 7th, 2014

Objectives

The lab on search and retrieval is split into two parts. The first part is about the vector space model (VSM) with TF-IDF weights and latent semantic indexing (LSI), an extension of VSM. The second part covers pLSI, a probabilistic model that automatically identifies different topics in a corpus, and assigns a distribution over topics to each of its documents.

This lab's objectives are:

- understand the idea behind modeling documents and how to search across the corpus with this model,
- understand the limitations of the vector space model, and how our favorite dimensionality reduction tool (SVD) can be taken advantage of to overcome these limitations,
- understand how to implement some algorithms used for search & retrieval using the MapReduce paradigm,
- understand pLSI and how Expectation-Maximization is used to optimizes the log-likelihood function.

Deliverables

- VSM
  - 20 search results for the query "James Bond 007"
- LSI
  - 20 search results for the query "James Bond 007"
  - The terms that have extremal values along the first dimension uncovered by the SVD
- pLSI
  - Plot of the log-likelihood at each iteration
  - Top 20 words for each topic
  - Latent meaning of each one of the topics
  - A way of adapting the clustering algorithm of the previous lab to automatically cluster documents based on their topic distribution

1 IMDB Movie Plots Dataset

The first dataset that we will explore this week contains plots written by users of the IMDb Web site[1] for about 200,000 movies. The data has been slightly pre-processed to make it easier to work with. You can take a look at it by opening `/ix/imdb-plots.txt` on HDFS. This file contains the movie plots together with the ID of the movie they belong to; `/ix/imdb-titles.txt` contains the mapping between movie IDs and title.

- Spend some time inspecting the raw data. Are you able to find your favorite movie in? What languages are the plots written in?

Also note that a small sample of the dataset is provided with the handout, in `data/imdb-sample.txt`. Use it to test your jobs locally before sending them to the cluster.

2 Vector Space Model

The first section focuses on one of the earlier models used to represent, manipulate and rank text documents, the *vector space model*.

Remark: This lab uses third-party Java libraries. In order to properly configure your environment (just like for one of the previous lab) we ask you to run the following command every time you connect to one of the cluster-access servers:

```
source scripts/setup.sh
```

The script is available in the archive you donwloaded from Moodle. Make sure that you *source* it, not just execute it. This will set up environment variables and configure aliases that will make your life easier.

An important assumption underlying the vector space model is that documents can be represented as the set of terms they contain (the well-known *bag of words* assumption). The question then arises of how to properly identify these terms given a stream of characters for a document. This almost inevitably leads to the need for pre-processing the data. We've already taken care of this for you; take a look at the file `DocumentTokenization` if you're interested. In particular, it contains a method that, given a line of text, returns a collection of *tokens* ready to be used as terms.

2.1 TF-IDF Weights

Let us start by creating the term-document matrix. The value of this matrix at index (t, d) indicates the importance (or relevance) of term t to the document d. For this purpose, we will use the TF-IDF weight for each pair (t, d) defined as $tf_{t,d} \times idf_t$, where the *term frequency* is a function of both term and document:

$$tf_{t,d} = \frac{\#\text{ occurrences of } t \text{ in } d}{\text{length of } d},$$

where the length of d is the total number of words in d, and the *inverse document frequency* is a function of the term only:

$$idf_t = \log\left(\frac{\text{total }\#\text{ of documents}}{\#\text{ of documents containing } t}\right).$$

In order to compute the TF-IDF weight for each term-document pair we will need a sequence of 3 MapReduce jobs:

[1] See: `http://www.imdb.com/`.

1. The first job will transform the raw lines in the dataset to key-value pairs `((term, docID), count)`, where `count` is the number of times `term` appears in `docID`.
2. The second job will append to each pair the length of the corresponding document, by summing the term counts for each document: `((term, docID), count, length)`.
3. Finally, the third job will count the number of documents containing each term, and output the TF-IDF weight for each term-document pair: `((term, docID), weight)`.

It is now time to start coding! The relevant java classes are located in the package `ix.lab07.vsm`. All the code that you will write can be checked with unit tests—make sure to take advantage of them.

- Let's start with something simple: complete the functions `termFrequency` and `inverseDocFrequency` in `TfIdf.java`. A note for later: executing `TfIdf` automatically sets up and launches the 3 MapReduce jobs described above.
- The first job is implemented in `WordCount.java`. Take a look at the Mapper, and complete the Reducer. It should remind you of the very first lab...
- The second job is in `DocumentLength.java`. Implement the Mapper, and try to understand how the Reducer works. Note the following: we need to write out the total number of terms for each value we get in the Reducer, but we know this only after having processed all the values. Therefore, we need to store all the values as we iterate over them and then make a second pass[2].
- The last job is in `WordWeights.java`. Implement the Reducer. Here as well, you will need to store the input values as you iterate over them in order to make a second pass.

Make sure the 3 jobs pass all unit tests, and run `TfIdf` locally by using the sample input file provided in `data/imdb-sample.txt`. Inspect the output of each of the jobs, and make sure you understand what's happening. When you are ready, export a JAR with your code and run it on the cluster:

```
hadoop jar ix-tlab07.jar ix.lab07.vsm.TfIdf -libjars $JARS /ix/imdb-plots.txt <output>
```

Don't forget the `-libjars` argument—it tells Hadoop to send additional libraries over to each machine in the cluster.

2.2 Search

Now that we've built the term-document matrix, let us use it to search for movies. The idea is as follows: consider the query string as just another vector (according to our model), and compute the *cosine similarity* (a normalized dot-product) of the query with each document; this similarity can be interpreted as a score. Finally, return the documents with the highest score.

The search is implemented in `Search.java`. As input, it takes a query string and the path to the folder containing the TF-IDF weights of each (t, d) pair—exactly what we just got. Computing the score for each document is done using a MapReduce job.

- Complete the Reducer in `Search.java`. It receives all the terms and the corresponding weights for a particular document, as well as the terms and weights of the query (through a configuration option) and outputs the score for the document with respect to the query. Read carefully the comments in the code, and test your code using the unit tests.

[2] You might wonder why we can't just use the `Iterable` Hadoop gives us twice. Simply put, it may contain *a lot* of values, such that it might be a good idea to discard them once processed. Hence, the framework leaves the caching up to us.

- Export a JAR archive, and run `Search` on the cluster with different queries (don't forget to set the `-libjars` option as before). For starters, try to search for "James Bond 007". What do you think of the results? Try out different queries.
- The `Search` command can also read the query string from `stdin` if you give a dash (`-`) as third argument. We included an excerpt of the Internet Analytics course description in `data/document.txt`. What movies do you get back when you use this description as a query string? Can you explain why "Babe (1981)" has such a high ranking?
- By providing a document as the query we get a simple content-based recommender system. On the IMDB website, find the plot of your favorite movie, save it into a text file and use it as a search query string. The results, i.e. the *nearest neighbors*, could then be considered as recommendations.

Tip: if you want to read the plot of one of the movies, you can `grep` the datasets. Suppose you want to read the plot for "Dr. No (1962)":

```
hadoop fs -cat /ix/imdb-titles.txt | grep "Dr. No (1962)"
# Lookup the ID of the document at the beginning of the line.
hadoop fs -cat /ix/imdb-plots.txt | grep "058242"
```

3 Latent Semantic Indexing

In the vector space model, we are limited by the fact that each term has its own dimension and is completely independent of the other terms. An issue of such an approach is (near-)synonymy: we would like documents including the term *clementine* to show up when the query contains the (slightly more general) term *tangerine*. To address that issue, we would need a technique that automatically uncovers the hidden relations between the terms... You guessed it, we will take advantage of our favorite dimensionality reduction tool, singular value decomposition. By applying SVD on the term-document matrix, we project both terms and documents onto a low-dimension latent space of *concepts*. In the context of information retrieval, this is known as *latent semantic indexing* (LSI).

- We first need to process the output of `TfIdf` and reshape it into a matrix whose rows are terms and documents are columns. We already coded this for you, simply run `TermDocumentMatrix` on the cluster (again, don't forget to set the `-libjars option`). How many rows and columns does the matrix have?
- We will now use an off-the-shelf SVD implementation provided by Mahout. Run the following on the cluster:

  ```
  mahout ssvd --input <input path> --output <output path> --tempDir <tmp> \
   --rank 200 \
   --oversampling 20 \
   --powerIter 1 \
   --reduceTasks 10
  ```

  This computes an (approximate) truncated SVD of rank 200, which should take a few minutes to run.
- Run the `Concepts` command on the cluster to get a feel of the concepts behind the first few dimensions uncovered by the transformation. The first dimension is particularly striking. How can you explain what you see?
- The LSI counterpart to `Search` is `LSISearch`. Run some queries and compare the results to those obtained using the vector space model. What do you think of the results? Are they

better, worse? Can you get an intuition about what's happening? *Note: no need to use the* `-libjars` *option this time as there is no MapReduce job.*

Remark: As in the previous lab on recommender systems, we often have to deal with *sequence files* which are encoded in a (standard) binary format defined by Hadoop. One of the main advantages of the format is that it makes it easier to serialize complex data types (e.g. vectors). However, if you open a sequence file, you will see only binary gibberish. To make sense of it, use the Hadoop command line utility:

```
hadoop fs -text path/to/seqfile
```

The `-text` option is very similar to `-cat`, except that it processes the binary data and outputs something that is readable by a human.

4 pLSI

LSI, as presented above, uses the SVD of the TF-IDF matrix to capture latent topics in a corpus. In this section, we will study probabilistic Latent Semantic Indexing (pLSI), that takes a probabilistic approach at the problem of capturing topics. In the remainder of the lab, we ill use the following nomenclature for the dataset:

- A *word* is defined as an item from a vocabulary, represented as $\{0, 1, \ldots, V-1\}$.
- A *document* is a sequence of N words denoted by $w = (w_1, w_2, \cdots, w_N)$, where w_n is the nth word in the sequence.
- A *corpus* is a collection of M documents denoted by $D = \{\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_M\}$.

4.1 Dataset

In this second part, we will use a dataset composed of 2150 news documents from Associated Press, stored in the file `data/associated-press.txt`. This file contains on each line a document id, followed by the content of the news item. An example of such a line is shown below:

```
21  The Nikkei Stock Average closed at 28,099.84 points, up 113.85 points, on the Tokyo
    Stock Exchange Tuesday.
```

As with the IMDb dataset, we pass the text of the news items through the `DocumentTokenization` class, that gives us a list of terms.

4.2 Model

Let us first briefly remind ourselves what is the model behind pLSI. Based on the graphical model shown in Figure 1, word w and document d are independent given the latent topic z. Therefore, pLSI models the joint probability of a document d and word w as

$$P(d, w) = P(d)P(w|d) = P(d)\sum_z P(w|z)P(z|d). \tag{1}$$

Consequently, the whole corpus D of documents is generated as

$$D = \prod_d \prod_w P(d, w)^{n(d,w)}$$

where $n(d, w)$ is the number of appearances of word w in the document d.

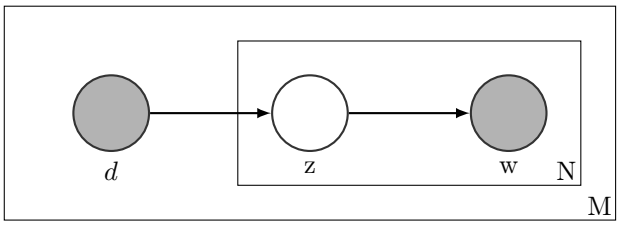

Figure 1: Graphical model describing pLSI

The log-likelihood of the whole corpus is defined as:

$$L = \sum_{d,w} n(d,w) \log P(d,w) = \sum_{d,w} n(d,w) \log \left[P(d) \sum_{z} P(w|z)P(z|d)\right]. \tag{2}$$

4.3 Expectation Maximization

The pLSI model is obtained by computing the values of $P(d), P(w|z)$ and $P(z|d)$ which maximize the log-likelihood function (2). To do so, we use an Expectation Maximization (EM) algorithm to maximize the value of L. It iterates over the two steps described below.

E-Step

The E-step updates the distribution of $P(z|d,w)$. The update rule for the expectation step is

$$P(z|d,w) = \frac{P(w|z)P(z|d)}{\sum_{z'} P(w|z')P(z'|d)}. \tag{3}$$

M-Step

In the maximization step, we update the words distribution per topic, and topic distribution per word, based on the distribution computed in the expectation step:

$$P(w|z) = \frac{\sum_{d} n(d,w)P(z|d,w)}{\sum_{d,w'} n(d,w')P(z|d,w')}, \tag{4}$$

$$P(z|d) = \frac{\sum_{w} n(d,w)P(z|d,w)}{\sum_{z',w} n(d,w)P(z'|d,w)}, \tag{5}$$

$$P(d) = \frac{n(d)}{\sum_{d'} n(d')}. \tag{6}$$

where $n(d)$ is total number of words (with repetition) in the document d. Note that $P(d)$ does not change across iterations, and can thus be computed once at the beginning.

Implementation

Our goal is to find the values of $P(d), P(z|d)$ and $P(w|z)$ which maximize the log-likelihood function (2). To do so, we provided you with a skeleton for the implementation of the EM algorithm in the class `ix.lab07.pLSI.PLSI`. The initialization of the different probabilities is given in the `initialize()` method. Have a look at it and try to understand how the different probabilities are represented and stored in the code.

- Implement the expectation step in the `eStep()` method of `PLSI`. This method updates the distribution of $P(z|d,w)$ based on Equation 3.

- Implement the maximization step in the `mStep()` method of `PLSI`. This method updates the word per topic distribution, and topic per document distribution, based on Equations 4 and 5. Note that the documents probabilities $P(d)$ are already computed in the initialization, and do not need to be updated.

We provide you with some basic unit tests that check that your distributions are correctly normalized. You can add some more advanced tests to check your implementation!

Results Interpretation

Now, it is time to run pLSI and interpret the results.

- Run `PLSI` on `data/associated-press.txt`. You can set the number of topics and iterations to 5 and 500 respectively. It should take 10-15 minutes to run (1-2 seconds per iteration). You can lower the number of iterations to debug your code.

The log-likelihoods of the corpus at each iteration are saved in the file `loglikelihood.txt`.

- Look at the evolution of the log-likelihood at different iterations of pLSI. You can use the script `scripts/loglikelihood.py` to plot the log-likelihoods at each iteration:

```
python scripts/loglikelihood.py output/loglikelihood.txt
```

- Do you think 500 iterations are enough? Why?
- How would you change the code to be sure that the algorithm has converged, without setting the number of iterations arbitrarily high?

The distribution of words for each topic are written to `Pw_z.txt`, with each row corresponding to a topic, and column to a word. We provided you with `scripts/topics.py`. that prints the top words of each topic.

- Look at the top 20 most probable terms for each topic and try to identify the meaning of each topic:

```
python scripts/topics.py output/Pw_z.txt output/vocabulary.txt 20
```

Along with the vocabulary and the word/topic distribution, pLSI also outputs the topics distribution per document in `Pz_d.txt`. We can use these probabilities to cluster documents. For example, we can assign each document to the topic that has the highest probability for this document. The script `scripts/cluster.py` implements this simple clustering algorithm.

- Run the clustering algorithm on the topics distributions you obtained and show 10 documents for each topic:

```
python scripts/cluster.py output/Pz_d.txt data/associated-press.txt <topic id> 10
```

- Look at the documents which are in the same topic. Do you observe any similarity between those documents?
- Are the topics of the documents in each cluster similar to the the meaning you found previously?

The script above used a "fixed" clustering, by simply considering the topic with the highest probability as the assignment of each document. However, we saw in the previous lab that it is possible to cluster documents automatically, using either hard assignments (k-means) or mixtures (GMM).

- How would you use the topic distribution of each document to automatically cluster them? If using k-means, do you thing that the Euclidean distance is a good metric to compute the "distance" between documents? What metric would you use?

If you have time, adapt your code from the previous lab to cluster documents based on their topic distributions, and visualize the clusters to see if they make sense.