Topian 0.1 Reference Manual

Emmanuel Eckard and Jean-Cdric Chappelier

17 September 2007

Abstract

This document describes Topian ("Topic-based Model layer for Xapian"), a software layer intended to add support for topical models to Xapian.

One common technical problem of Information Retrieval engines is to hold and access to data. We choose to use Xapian, a performent retrieval library focusing on probabilistic retrieval.

Modern Information Retrieval does not limit itself at representing documents in a vector space via indexing. A transformation of the vector space can take place, often involving a reduction in dimensionality. Starting with the Latent Semantic Analysis (LSI) model, a number fof models have been developed in the past 10 years, including Probabilistic Latent Semantic Analysis (PLSI), Naive Bayesian models, or Latent Dirichlet allocation.

We attempt at providing a general software framework to develop such models on the stable, fast and generally efficient basis of Xapian.

This document describes Topian ("Topic-based Model layer for Xapian"), a software layer intended to add support for topical models to Xapian.

1 Conception

Topian provides two trends to objects: objects directly linked to the documents, and abstract objects dealing with the model. Document representation is implemented in `DocsRep` objects which encapsulate Xapian objects like `Xapian::Database`. The `DocsRep` is what makes the link between the classical document representation, and features of the intended models held in `Parameters` objects. The `Parameters` are abstract objects which represent the model descriping statistical features of the document topics and word topics. This architecture keeps the `Parameters` distinct and separated from the `DocsRep`, allowing for selective loading of the data when both direct data and model data are not needed for a given process.

For instance, the PLSA model is a statistical model based on the parameters $P(w|z)$ (probability that, given a certain topic z, work w occures), $P(d|z)$ (probability that, given a certain topic z, document d occures), and $P(z)$ (probability that a topic z occures). On Topian, this translates into a `PLSARep` class inheriting from `DocsRep` data-wise, and a `PLSAParams` inheriting from `Parameters`, model-wise. The classes `PLSARep` and `Parameters` have further children to take specificities of queries versus collection documents into account.

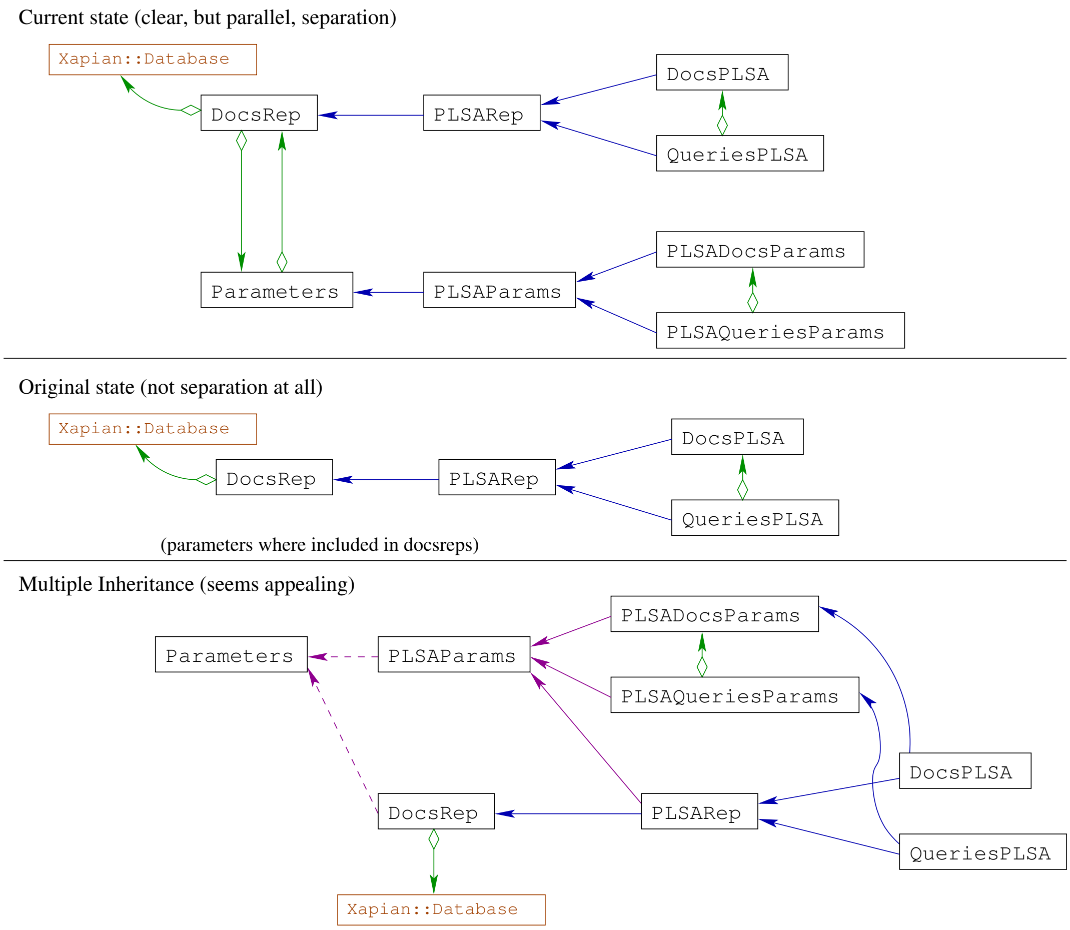

Figure 1: Conception of the Topian layer for PLSA

`TREC::DocId`	foobar, foo, bar, foo2
`Xapian::DocId`	1, 2, 23, 42
`Topian::DocId`	0, 1, 2, 3

Figure 2: Example of document identifiers

2 Document identifiers

Document identifiers in NLP toolchains can take several types: `int`, `string`, etc. In the concrete case of using Xapian and Topian to process TREC data, we find ourselves with three flavours of identifiers. TREC identifiers are arbitrary strings; Xapian identifiers, typed `Xapian::DocId` are non-continuous integers starting from 1, 0 being reserved for out-of-collection documents; and Topian uses continuous integers starting with 0. We will name these three types of identifiers `TREC::DocId`, `Xapian::DocId` and `Topian::DocId` (see figure 2).

Xapian documents have a string container whose content can be accessed from `Xapian::Document::get_data()` member. The `Xapian::DocId` of a document can be retrieved either from the document itself with the `Xapian::Document::get_docid()` member, or by dereferencing an iterator over the `Xapian::Documents` with the `operator*() const`.

Over the course of the processing, we switch from the TREC identifiers to internal identifiers, and back again. This allows abstracting the TREC format if necessary (for instance when using the software on another format), and using more efficient algorithms that maps of strings.

Two standards containers of the `std` library could be used to map different types of docids: maps and vectors. Vectors map continuous integer indices to a type. They have an access time in $O(1)$. Maps allow using any type as indice; in particular, they allow using strings or non-continuous integers as indices. Their access time is in $O(\log n)$. Using an iterator, going through a whole container (map or vector) is $O(n)$. Hence it is adventageous to use vectors for punctual accesses, but performance is not lost if a whole map is processed linearly.

We propose to implement the translations between the formats as follow:

- Xapian to Topian: the corresponding `Topian::DocId` is stored in a `std::map` contained in `DocsRep`. This data *must not* be contained in the `data` attribute of the `Xapian::Documents` because we need it to contain the `TREC::DocId`, which is set there at indexing time. Furthermore, `data` is of the `std::string` type, and it is unclean to set Topian things in the Xapian database.
- Topian to Xapian: a `std::vector` of `Xapian::DocId` is created.

`Topian::DocID`	0	1	2	3	...
`Xapian::DocID`	1	2	23	42	...

  Access being is $O(n)$, this structure can be used for punctual access (by opposition to skimming through the whole collection)
- Topian to TREC: a `std::vector` of `TREC::DocId` is created. This vector can simply be constructed at indexing time, since the `Xapian::DocId` indexing scheme is trivial.

`Topian::DocID`	0	1	2	3	...
`TREC::DocID`	foobar	foo	bar	foo2	...

This makes it possible to access the `TREC::DocId` of any document in $O(1)$.

It is straigthfowards that Xapian to TREC is obtained through Topian, also in $O(1)$. With the orientation of the data flow, it is not necessary to translate `TREC::DocId` into the internal indices `Xapian::DocId` and `Topian::DocId` (which should stay transparent TREC-wise anyway). Hence, we have a whole set of translators.